TIMING-SAFE HARDWARE-LEVEL INFORMATION FLOW CONTROL

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Andrew Ferraiuolo

May 2018

TIMING-SAFE HARDWARE-LEVEL INFORMATION FLOW CONTROL

Andrew Ferraiuolo, Ph.D.

Cornell University 2018

Developing secure processors has become increasingly important. Recent advancements in commercial security architectures such as Intel SGX have garnered much attention. The promise of these architectures is compelling; because the root of trust lies in hardware, the system it protects remains secure even if low-level software such as the operating system is compromised. Unfortunately, hardware is itself complex and error-prone – software-exploitable vulnerabilities in SGX have already been found, and bugs in hardware have a long history of causing security vulnerabilities. Even if hardware is correct in the conventional sense that it conforms to a specification, security is not assured. Processors still leak secrets through microarchitectural timing channels, and virtually every hardware feature that improves performance has been exploited to leak secrets.

Information flow security is a promising approach for verifying hardware systems. Information flow tracks and constrains the movement of data throughout a system ensuring that confidentiality and integrity are not violated. Information flow control can be enforced by a type system embedded in a secure hardware description language (HDL) with which the hardware is described. With this approach, the HDL includes features for describing security policies. The type system of the HDL then statically proves at design-time that the security policies are always enforced.

This thesis enables future hardware designs to provide strong assurance

through information flow control. This is achieved through two major thrusts of research: 1) developing a practical, expressive hardware description language for enforcing information flow control in hardware designs and 2) designing secure hardware that is suitable for information flow verification. In particular, this thesis describes a number of contributions to secure HDLs that were implemented as extensions to a secure HDL called SecVerilog. It also describes ChiselFlow, a variant of the HDL, Chisel, for information flow security. This thesis studies the application of secure HDLs to a prototype implementation of a commercial security architecture, ARM TrustZone. A novel architecture for information flow security called HyperFlow is presented to improve upon the shortcomings of TrustZone by providing more general security policies, constraining information release, and defending against timing-channel attacks. A second novel architecture called Timing Compartments extends HyperFlow to defend against timing-channel attacks in a multicore processor. A novel memory scheduling algorithm for preventing timing channel attacks in a shared memory controller is presented to improve performance. Overall, this thesis demonstrates that HDL-level information flow control is capable of securing usable and performant hardware designs.

BIOGRAPHICAL SKETCH

Andrew Ferraiuolo received Bachelors of Science degrees in Computer Engineering and Electrical Engineering (Magna Cum Laude) from the University of Connecticut in 2012. Shortly afterwards, he began the PhD program in the department of Electrical and Computer Engineering at Cornell University where he is advised by G. Edward Suh and works closely with Andrew C. Myers. He has interned with Microsoft Research Labs in Redmond, Washington. His research lies at the intersection of computer security, computer architecture, and programming languages.

ACKNOWLEDGEMENTS

I would not have been able to complete a PhD or this thesis without the support of many people. I have been fortunate to have had many allies in both technical and non-technical capacities during my academic pursuits.

First, I would like to thank my advisor G. Edward Suh, and my special committee member Andrew C. Myers, whom I have also worked with closely. Without the guidance, mentorship, and insight of both of them, the research described in this thesis would not have been possible. It has been an honor, privilege, and pleasure to have worked with and known both of them. I would also like to thank my special committee member Zhiru Zhang for his thoughtful suggestions and feedback on this research.

I would also like to thank the many other PhD students with whom I have had the pleasure of working with and have contributed significantly to the research in this thesis. First I would like to thank Danfeng Zhang. Danfeng's work on SecVerilog has inspired me to work on securing hardware with information flow control, and I am grateful to have had the opportunity to build upon his research. I am also grateful for his guidance and feedback while obtaining formal results for extensions to SecVerilog. I would also like to thank Rui (Chris) Xu for his effort on the TrustZone prototype processor. Yao Wang also contributed to the work on timing compartments. I would like to thank Weizhe (Will) Hua for writing the prototype pipeline for the work on mutable dependent information flow labels. I would like to thank Yuqi (Mark) Zhao for test-driving an early version of ChiselFlow and helping to label HyperFlow.

I would also like to thank my mentor and collaborators from my internship at Microsoft Research, Andrew Baumann, Bryan Parno, and Chris Hawblitzel. I am grateful to have been a part of the Komodo Project.

The Suh Research Group and Computer Systems Lab have both been wonderful research communities to have been a part of. The many trips with CSL students for lunch in collegetown have been a pleasure. I would like to thank Dan Lo for his mentorship as a senior PhD student and friendship. Dan has been great to bounce ideas off of and provided guidance about how to navigate the PhD program. I am also grateful for the many hours not-wasted playing Netrunner with Dan and Jon Tse. I am glad to have been the benefactor of Shreesha Srinath's expansive knowledge of both computer architecture and new-and-exciting lunch spots. Our many walks for coffee have helped me through the most difficult moments over the past few years. I have enjoyed many pleasant and insightful talks about HDLs with Derek Lockhart. Tayyar Rzayev found a seemingly limitless number of fun things to do from rock climbing to space music. I am glad to have played many rounds of Dungeons and Dragons and other board games with Ji Kim, Chris Torng, KK Yu, Dan Lo, Berkin Ilbeyi, Kyle Wecker, Jon Tse, and Erik Halberg.

I would like to thank the other wonderful friends I have made during my PhD. It has been a privilege to have met Joshua Barrom, Samuel Kurland, Jen Keefe, Carlos Higgins, Preslava Staneva, Aaron Gittleman, Carlos Diaz, Joren Lauwers, and Austin Henley. Knowing each of them has made my journey much more enjoyable.

Finally, I would like to thank my family that has supported me during my academic pursuits and life. My sister Amy and mother have always been caring and encouraging without exception, and for that I am grateful.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Hardware plays an important role in ensuring that computing systems are secure because they provide security mechanisms such as protection rings, virtual memory support, secure boot, and memory encryption. A more recent trend among hardware vendors is to provide instruction set extensions that enable a software module to execute on remote, distrusted machines while approximating the security of executing that same software on a trusted machine [17, 8]. Security extensions such as these protect the trusted software from attacks by a malicious or compromised operating system, and provide attestation to allow the software to prove that its code and data are trustworthy to remote parties. These architectures are compelling because they approximate the security provided by cryptographic algorithms that are computationally prohibitive such as fully homomorphic encryption.

Unfortunately, microprocessors often contain vulnerabilities that allow security-critical software to be compromised by untrusted software. Software-exploitable vulnerabilities in SGX have already been found [62]. Previous studies found vulnerabilities in implementations of Intel VT-d [98] and system management mode (SMM) [97]. Moreover, the recent Spectre [42] and Meltdown [52] vulnerabilities show that even if the hardware is correct in a conventional sense—that it implements a specification—is not sufficient to ensure security. Exploiting subtle timing channels in Intel microprocessors, Meltdown can be used to leak arbitrary kernel data. Therefore, it is important for security to ensure that hardware implementations are free of timing channels.

Constructing hardware with a security-typed hardware description lan-

guage (HDL) [51, 50, 110] can provide strong security assurance. Security-typed HDLs can statically ensure that the hardware prevents insecure information flows: untrusted signals cannot affect trusted signals, and secret signals cannot affect public ones.

Security-typed HDLs are also lightweight. The hardware designer annotates the code with labels to describe a security policy that hardware should enforce, and the type system statically checks that this policy is implemented. Type checking is fast and takes place at design time. As a result, it has negligible impact on chip area, run-time performance, and power consumption.

Information flow type systems can offer formal guarantees, commonly *noninterference*. If H is a more restrictive information flow label then L, then noninterference prevents all flows from H to L. Noninterference can enforce both confidentiality policies and integrity policies. *Confidentiality* policies prevent flows from `secret` values to `public` ones, whereas *integrity* policies prevent flows from `untrusted` values to `trusted` ones.

Because HDLs give cycle-level descriptions of hardware, HDLs can enforce a particularly strong variant of noninterference that precludes timing channels. Noninterference is typically formulated as a set of pairs of traces. If a system begins executing from states s_1, s_2 that agree on values labeled L, and execution from s_1 and s_2 emits traces t_1 and t_2 respectively, then the system is noninterfering if those traces appear indistinguishable to an adversary that can view L values. Often, formulations of noninterference model an adversary that is incapable of timing measurements, for example, by giving a definition of "indistinguishable" that permits traces to differ in the number of consecutive states that agree on L values. These formulations of noninterference are *timing insen-*

sitive because they do not distinguish the timing of events. As a result, timing insensitive formulations of noninterference do not prevent timing channels. By contrast, *timing sensitive* definitions of noninterference defend against timing channel attacks and model adversaries that can measure timing. Timing insensitive definitions of noninterference are used in most systems for information flow control because controlling timing channels is difficult. Prior work has shown that information flow control HDLs can enforce a timing sensitive variant of noninterference [51, 50, 110].

1.1 Contributions and Outline

This dissertation makes contributions to both secure hardware description languages and to hardware architectures that defend against timing-channel attacks and can be statically type-checked with a secure hardware description language.

Chapter 2 gives background information on secure hardware description languages and then describes the contributions of this dissertation to secure HDLs. The novel contributions include support for heterogeneously labeled data structures, purely statically checked dependent labels, controlled downgrades of information flow policies, label inference, and information flow control embedded in a higher-level language. Many of these features are implemented as extensions to the previously proposed secure HDL, SecVerilog.

This dissertation also describes a new secure HDL called ChiselFlow. ChiselFlow is the first secure HDL to provide label inference. Label inference automatically deduces the labels of signals internal to a hardware design. As a

result, only ports must be labeled explicitly, so the annotation effort required by the hardware designer is reduced. ChiselFlow is implemented as an extension to the embedded DSL for hardware design, Chisel. Chisel is embedded in a higher-level programming language called Scala, granting it much of the expressive power of Scala. By extension, ChiselFlow is also expressive, and it is the first information flow control language to support the synthesis of hardware from a higher-level language.

This dissertation also presents a way to enforce policies described by dependent labels fully statically. *Dependent labels* are information flow labels that depend on the runtime values of variables. Dependent labels can be used to encode run-time properties of a system as information flow policies. They are especially important for hardware because they are a way to describe hardware that can be shared at run-time between different security domains. In this way, dependent labels allow hardware to be partitioned both spatially and temporally. However, applying dependent labels to temporally partitioned hardware can introduce subtle vulnerabilities if not handled carefully, because dependent labels can change at run time. Prior secure HDLs have supported dependent labels by introducing dynamic checks that convert possible violations of security into possible functional correctness errors [51, 110]. However, dynamic checks prevent some common and important hardware designs, and necessitate extensive functional testing to ensure that these dynamic checks have not created functional errors. The type system presented in this thesis is both expressive enough to permit practical hardware designs and enforces security statically, reducing the testing burden. This thesis also provides a formal security result that shows that the type system with fully statically-enforced dependent labels and heterogeneously-labeled data structures enforces a timing-safe variant of

noninterference.

Languages for information flow control often intend to enforce noninterference [74]. However, noninterference is too restrictive for practical systems. Information computed using secrets may eventually need to be released to the public and untrusted inputs may be permitted to influence trusted state, for example, after the inputs have been sanitized. As a result, all practical systems for information flow control support *downgrading* which relaxes information flow policies. Downgrading that relaxes confidentiality policies is said to *declassify* whereas downgrading that relaxes integrity is said to *endorse*. Because downgrading weakens noninterference, prior work has examined downgrading systems to limit the extent to which they might cause harm. This thesis applies prior approaches to controlling downgrades that allow a useful weakening of noninterference to be proved. In particular, it applies *robust declassification* [101], which prevents untrusted parties from influencing declassification. It also applies *transparent endorsement* [10] which only allows a party to endorse information that it could have read. This thesis is the first to support downgrades in a secure HDL. There are also few systems built with constrained downgrades, and by securing a processor with a system that constrains downgrades, this thesis provides a data point that suggests that robust declassification and transparent endorsement are expressive enough to permit practical systems while still ensuring security.

Chapter 3 applies the extended version of SecVerilog to statically checking the information flow security of a prototype implementation of ARM TrustZone. TrustZone is a commercial security architecture that partitions the hardware into two trust domains: the secure world, and the normal world. This

study shows that static information flow security is applicable to practical security architectures. The results in terms of area overhead and programmer effort suggest that static information flow security is lightweight. Case studies in which we replicate real-world vulnerabilities found in real processors suggest that static information flow security also catches real hardware bugs. Our TrustZone-like prototype is also the first information-flow secured processor to support the simultaneous protection of both confidentiality and integrity. Prior information flow secured processors [87, 85, 86, 51, 50, 110] provide two hierarchical security domains. As a result, they can protect just one of confidentiality or integrity. The TrustZone-like prototype provides a confidential and trusted domain that is incomparable with a public and untrusted domain. However, because the threat model of TrustZone does not consider timing channels, our prototype implementation does not defend against timing attacks. The prototype also relies on unconstrained downgrades to permit communication between worlds.

Chapter 4 presents a novel, secure hardware architecture for information flow security called HyperFlow. HyperFlow generalizes conventional, purely hierarchical notions of privilege with lattice model information flow policies, and it provides memory protection through tags that also represent information flow policies. Enforcing lattice model policies in hardware has a number of advantages.

Providing security with physically tagged memory pages both reduces the software trusted computing base and makes the implementation of HyperFlow more amenable to static checking with an information-flow-secure HDL. Virtual memory does not in general enforce noninterference because virtual pages

in the address spaces of mutually distrusting processes can be mapped to the same physical page. Further, because the page tables are software-defined, the security of a particular system configuration cannot be determined purely by inspecting the hardware implementation.

Because the memory tags in HyperFlow are information flow labels, they are capable of expressing rich application security policies. Prior work on operating systems [24, 14, 104] and programming languages [66] for information flow control suggest that applications require complex security policies that support communication among mutually distrusting principals. For example, it is often necessary for an actor, Alice, to send a piece of information to another actor, Bob, that Alice does not trust. Alice might need Bob to perform some computation on her data, but Alice does not want Bob to be able to send her data elsewhere. Prior work on decentralized information flow control (DIFC) is capable of enforcing security policies of this form. Exposing DIFC application security policies to the hardware simultaneously supports more precise separation of privilege and provides potential performance benefits to the operating system because it can reduce the number of privilege changes.

In order to support incremental adoption of protection using information flow labels, HyperFlow provides a hybrid protection model in which both conventional virtualization-based protection and protection with information flow labels are supported simultaneously. By supporting both memory protection models, HyperFlow supports systems in which some applications and libraries are modified to take advantage of protection with information flow labels, whereas unmodified legacy applications can still rely on virtual memory. By providing both defense mechanisms, HyperFlow also supports the use of

virtual memory for reasons other than security. For example, virtual memory provides a simple mechanism to relocate programs in memory.

HyperFlow improves upon processors which have been information-flow-secured by supporting communication across security domains in memory for IPC and through registers for system calls and function libraries. Such communication is enabled securely through the ISA which includes constrained downgrading instructions. Downgrading instructions only permit robust and transparent downgrading. By ensuring that downgrades are robust and transparent, HyperFlow can enforce DIFC policies.

HyperFlow also permits control transfers across security domains which is necessary for system calls and function libraries. Such control transfers are supported through a control gate mechanism [76, 96]. Control gates in HyperFlow tightly couple an entry point (pc value) with an information flow label that denotes the privilege of the code at that entry point.

Chapter 5 presents the Timing Compartments architecture. Timing Compartments provide timing channel protection among distrusting domains that share hardware resources in a multicore processor. By relying on simple protection mechanisms such as spatial and temporal partitioning, timing compartments provide strict noninterference, and are amenable to static checking with an information flow type system. Performance evaluations in a simulation environment suggest that straightforward temporal and spatial isolation incurs moderate performance overhead. We propose two optimizations that improve the performance compared to straightforward partitioning. We also identify that memory controller timing channel protection is the performance bottleneck.

Chapter 6 presents lattice priority scheduling which improves upon timing channel protection for memory controllers even further by taking advantage of memory transactions that are governed by lattice model security policies such as those in the HyperFlow architecture. Our observation is that prior memory controller protection techniques assume that all security domains are mutually distrusting. However, in practice, systems for information flow control have more complex trust relationships. In some cases it is possible for one transaction to delay another without violating the security policy. By more precisely enforcing these trust relationships, lattice priority scheduling both improves the total available memory bandwidth, and can more readily respond to the run-time behavior of applications to improve performance.

Overall, this thesis demonstrates that HDL-level information flow control is a practical approach to ensuring that hardware implementations are secure. Through the design and implementation of architectures for information flow security, this thesis also shows how to construct hardware that provides strong assurance and eliminates timing channels while still providing good performance. The contributions of this dissertation to HDLs for information flow security include:

- Support for heterogeneously-labeled data structures including arrays, bit vectors, and records.
- The first type system that supports dynamic information flow labels that are checked fully-statically at design time
- The first hardware description language with downgrades. In addition, downgrades are constrained so that declassification is robust [101] and endorsement is transparent [10]. Recent work has shown that a software

language which constrains downgrades to be robust and transparent can enforce a weakening of noninterference called non-malleable information flow control (NMIFC) [10].

- The first hardware design language for information flow control to support label inference.

The contributions of this dissertation to the design of secure hardware include:

- The TrustZone-like prototype is the first information-flow secured processor to protect both confidentiality and integrity simultaneously. By replicating security vulnerabilities found in real processors, we show that HDLs for information flow security can prevent real attacks.
- HyperFlow is the first hardware architecture to generalize memory protection and privilege in hardware to general, lattice-model information flow labels. HyperFlow is also the first information-flow secured processor to support communication through downgrades that are initiated by untrusted and public applications. Such communication is necessary to support IPC and system calls. HyperFlow also includes more performance-enhancing features than prior information flow secured processors. The inclusion of these features is interesting because conventional, insecure implementations of them would have timing channels.
- Timing Compartments is the first hardware architecture that prevents timing channels among concurrently executing processes that share the hardware in a multicore. Timing Compartments also proposes several performance optimizations.
- Lattice Priority Scheduling improves upon the performance of a memory controller for timing channel protection. LPS is the first timing-channel

free memory scheduling algorithm to enforce lattice model security policies precisely.

The work presented in Chapter 2 is adapted from [29] and [26]. The work in Chapter 3 is adapted from [29]. HyperFlow, presented in Chapter 4, is joint work with Yuqi Zhao, Andrew C. Myers, and G. Edward Suh. Chapter 5 is adapted from [27], and Chapter 6 is adapted from [28].

CHAPTER 2

HARDWARE DESCRIPTION LANGUAGE BASED INFORMATION FLOW SECURITY

Implementing hardware with security-typed hardware description languages (HDLs) is a promising approach to ensuring the hardware is secure. HDL-level information flow control applies techniques from language-based security [74] to hardware design [110, 50, 51, 26]. Variables in the code that describe the hardware design are annotated with security labels, L, which are types that describe restrictions on where information contained in that signal can flow. The type system then enforces these restrictions. This thesis improves upon the expressiveness of HDLs for information flow control while ensuring that security is preserved.

Type systems for information flow security can enforce noninterference [33], which ensures that a signal with a label L can only be influenced by signals with labels that are less restrictive than L. For example, if the label `public` is less restrictive than the label `secret`, then a `secret` signal cannot influence a `public` signal. This describes a policy that governs confidentiality because it prevents secrets from being leaked to the public. Dually, type systems for information flow control can enforce integrity policies by preventing untrusted signals from influencing trusted signals.

HDLs for information flow security can enforce a particularly strong, timing-safe variation of noninterference [110]. HDLs describe hardware at the register transfer level (RTL) – the code describes new valuations of signals during each clock cycle. Because HDLs give cycle-level descriptions of hardware, the information flow type system can guarantee cycle-level timing-channel freedom.

```
1  reg [31:0] {T} creg, [31:0] {U} untr, [31:0] {T} trst;
2  ...
3      creg <= untr; // not allowed
4      creg <= trst; // allowed
5  ...
6  reg {T} mode;
7  // mode_to_lb(0) = T, mode_to_lb(1) = U
8  reg [31:0] {mode_to_lb(mode)} gpr;
9  ...
10 if (mode == 1'b0) creg <= gpr;
11 ...
```

Figure 2.1: SecVerilog code example.

Much of the work in this chapter describes work that was implemented as extensions to SecVerilog, a variant of Verilog for information flow security. Verilog is a widely used language for hardware design. This thesis also develops a new variant of Chisel for information flow control called ChiselFlow. Chisel is an HDL embedded in Scala. ChiselFlow extends Chisel [4], an HDL embedded in Scala. An advantage of the ChiselFlow implementation is that it gains much of the expressiveness of Scala. Chisel emits a simpler, compiled intermediate representation called FIRRTL, which can then be used to produce hardware designs. ChiselFlow emits Secure Intermediate Representation for RTL (SIRRTL), which extends FIRRTL. The enforcement mechanisms of ChiselFlow operate entirely on SIRRTL. Many of the same techniques are applicable to both languages. As a result, this thesis discusses these language features using examples from both ChiselFlow, and an improved version of SecVerilog that this thesis refers to as SecVerilog2.

2.1 Fully-statically checked, dynamic security labels

Many practical hardware designs cannot be implemented efficiently with purely static labels. Labeling a component `T` means it can only be used ex-

clusively by one security level. If the same functionality were needed for another security level, the hardware module would have to be duplicated. To design efficient hardware, it is essential that hardware resources can be shared among multiple security levels over time. In SecVerilog [110], sharing is permitted through *dependent types* that express functions of free variables in the description of the hardware module, such as the label of `gpr` on line 10. This label, `mode_to_lb(mode)`, is a function of the signal `mode`; the label is `T` when the `mode` bit is `0` and `U` otherwise. Even though the label of `gpr` depends on the run-time value of `mode`, the assignment on line 10 can still be type-checked statically using a static program analysis. Because the assignment happens under a branch in which `mode` is `0`, the program analysis can infer that the label of `gpr` is `mode_to_lb(0) = T` in this context.

Unfortunately, dependent types introduce subtle security vulnerabilities when the variables on which they depend can change. Figure 2.2 illustrates this problem. The signal `shared` has dependent label `mode_to_lb(v)`, `v` is trusted, and `trst` and `untr` are labeled in the same way as before. This code is clearly insecure; on line 5, an untrusted value is stored in the shared register and this untrusted value is directly copied into the trusted variable on line 6. These lines can be executed in sequence over two clock cycles. This type of leakage through changes in dependent types is known as *implicit downgrading* [110]. In information flow control, secret information may be explicitly *downgraded*, i.e., released to the public, when the designer deems this release necessary and secure [75]. However, when downgrading is implicit, it represents a potential security vulnerability that the designer is unaware of.

Prior hardware languages for information flow control that support depen-

```
1  // mode_to_lb(0) = T, mode_to_lb(1) = U
2  reg {T} v, {T} trst, {U} untr;
3  reg {mode_to_lb(v)} shared;
4  ...
5      if (v == 1'b1) shared <= untrusted;
6      else           trusted <= shared;
7  ...
```

Figure 2.2: Implicit downgrading example.

dent types prevent implicit downgrading through dynamic code transformations [50, 110] that convert harmful downgrades of this form into possible functional errors. The code transformations used by SecVerilog are called *dynamic clearing* [110] — in this approach, the compiler inserts logic to clear dependently labeled registers whenever the labels of these registers are changed [109]. Dynamic clearing has severe practical limitations. It can cause hard-to-detect functional errors because it adds extra logic in the background that is not specified in the code. The added clearing logic causes the simulations and synthesized hardware to differ from what the designer would expect.

Dynamic clearing also makes it impossible to describe many hardware designs. We illustrate these limitations by describing the complications that dynamic clearing causes for the design of a widely-used processor feature — a privileged kernel mode and a user mode. Naturally, the labels for many processor resources will depend on the control register that indicates the current mode. General purpose registers (GPRs) should have labels that depend on the mode, since the trustworthiness of their contents depends on the mode that wrote them last. The program counter (`pc`) will also have a label that depends on the mode. Pipeline registers should have the same mode-dependent labels to reflect the privilege level of in-flight instructions. When the mode switches from user (`U`) to privileged (`T`), all of these registers would be dynamically cleared—whether the hardware designer wants this behavior or not.

Dynamic clearing prevents legitimate communication between security levels. For example, a system call instruction in the above processor example will trigger a label change from `U` (user) to `T` (privileged). Typically, some of the GPRs are used to pass information such as a system call number or arguments from the user mode to the privileged supervisor mode. Automatically clearing the GPRs during this mode switch breaks the functionality of system calls. Instead, the secure design language should allow the designer to *explicitly* downgrade the label of a register in certain cases so that its value can be preserved on a label change.

Here, and in other cases, dynamic clearing damages integrity. If the pipeline registers were automatically cleared on a mode change, in-flight instructions would likely be converted erroneously into NOPs. More generally, dynamic clearing can remove secrets and protect confidentiality, but when a trusted register is expected to contain a specific value, replacing it with a zero violates integrity. Ideally, the security type system must be precise enough so that it only requires explicit handling of label changes only if necessary for security. Dynamic clearing conservatively erases data on *any* label change. For example, a label switch from `T` to `U` on a return from a system call is not a concern for integrity; restoring a PC value from a saved one (in the epc register in MIPS) should not require explicit downgrading.

The type system described here is expressive enough to describe all of the above hardware while securely avoiding the implicit downgrading proble. Communication among security levels in the GPRs is permitted by explicit downgrading. The type system precisely tracks the direction of label changes allowing our design to load a trusted value into the pc on entry to the kernel,

```
1  wire com {T} mode_switch;
2  assign mode_switch = decode_out[4];
3
4  reg seq {U} epc;
5  reg seq {T} mode;
6  reg seq {mode_to_lb(mode)} pc;
7  // mode_to_lb(0) = T, mode_to_lb(1) = U
8  always@(seq) begin
9      if (rst) pc <= 16'b0;
10     else if (mode_switch && (next mode == 1'b0))
11     pc <= `SYSCALL_PC_VAL; //switch to kernel mode
12     else if (mode_switch)
13         pc <= epc; //return to user mode
14     ...
15 end
```

Figure 2.3: PC during mode switches.

and restore a saved pc on re-entry to userspace. The new type system permits design choices. For example, we can think of two correct implementations of mode switching: 1) pipeline the labels along with the regular pipeline registers, and 2) stall the pipeline until all in-flight instructions are drained. Our type system supports both designs.

Our type system securely supports changes in dependent labels 1) by making the propagation of signals on clock edges explicit in the syntax, semantics, and type system, 2) by introducing a syntax for testing labels for the next clock cycle, and 3) by using the type system to statically establish that registers are securely updated along with their labels. Figure 2.3 shows code for a PC register that securely handles mode changes in the concrete syntax of SecVerilog2.

Notably, only sequential logic can be implicitly downgraded through label changes since combinational logic is not stateful [110]. For this reason, combinational and sequential logic are separated in the language. In SecVerilog2, sequential and combinational variables are explicitly separated through type annotations `com` (on line 1) and `seq` (on lines 4–6). Sequential and combinational signals are type-checked differently. For example, an assignment to a trusted

combinational signal such as `mode_switch` defined on line 2 is secure as long as the value that is assigned is also trusted.

However, sequential signals (registers) such as `pc`, are type-checked in a different way. The values assigned to registers must be type-checked based on the *new* label of the register for the *next clock cycle*. This ensures that the new label of the register accurately reflects the security level of its contents. As an example, for the assignment on line 11 to type-check, the type system must prove that the label of `'SYSCALL_PC_VAL` is permitted to flow into the new label of `pc`, which is dependent upon the value of `mode` in the *next* clock cycle. In the example, the label can be statically determined to be `mode_to_lb(0)` (`T`) as we explain in the following paragraph.

SecVerilog2 supports a new operator, `next`, which when applied to a variable, gives the value it will take during the *next* cycle. For example, it is applied to `mode` on line 10, where it evaluates to the value of `mode` during the next cycle. The branch on line 10 is taken when there is a mode switch and the next-cycle value of `mode` is `0`, indicating a switch to kernel mode. The type system can thus infer that on line 11, the label of `pc` during the next cycle is `T`, and the assignment is safe as long as `'SYSCALL_PC_VAL` is trusted. On line 13, the branch was not taken, so the next-cycle label of `pc` must be `U`, and the assignment is safe. When the `next` operator is applied to registers (declared `seq`), the next-cycle value is available from the combinational input to the register. The next operator is useful both for implementing necessary access controls and assisting the type system.

Since type checking depends on whether variables are sequential or combinational, the syntax and semantics of SecVerilog2 ensure that the `com/seq` labels

are accurate (i.e., that variables labeled `com` are not sequential). In SecVerilog2, the clock signal is implicit, and sequential logic is written by describing its combinational input, as is done on lines 8–15. In the semantics, the statements describing sequential logic are treated as the combinational input to a register. Restrictions are then placed on combinational wires which ensure that they are in fact combinational: 1) there are no combinational loops, and 2) there are no inferred latches. A hardware module contains a combinational loop when there is a cycle in the logic that defines a wire, and the cycle does not include a register. A module includes an inferred latch when there is a condition under which a new value for a combinational signal is not defined.

In ChiselFlow, differentiating between sequential and combinational variables is straightforward. In ChiselFlow, clock signals are implicit, and assignments to variables that are declared as registers describe the logic at the input to the register that is latched-in on each clock cycle. The FIRRTL compiler prevents wires from describing combinational loops or inferred latches.

2.2 Heterogeneously labeled data structures

2.2.1 Bit vector types

Hardware designs often use sequences of bits to describe data structures. For example, one might construct a packet as a collection of bits describing data, an address, and possibly other metadata. SecVerilog reasons imprecisely about individual fields of a packet, since the whole packet must share a single label. Information about the labels of individual wires is lost once they are grouped.

```
1  wire [0:31] {world(ns)} data;
2  wire [32:41] {PT} addr;
3  wire {PT} ns;
4  wire [0:42] {i -> if (i <= 31) world(ns) PT} packet;
5  assign packet = {ns, addr, data};
```

Figure 2.4: A packet concatenation example.

Kinds	$k ::= \ell \mid \texttt{int} \rightharpoonup k$
Types	$\tau ::= \ell \mid \tau_1 \sqcup \tau_2 \mid \tau_1 \sqcup \tau_2 \mid x \mapsto \tau \mid fx$
	$\mid \mathit{if}\ e^{\tau}\ \tau_t\ \tau_f \mid \mathit{case}\ e^{\tau} \tau_1 \ldots \tau_n$

Figure 2.5: Extended Type Syntax for Arrays and Bit Vectors.

Figure 2.4 shows an example that creates a packet by concatenating data with an address a bit, `ns`, that describes whether the data belongs to a public or confidential security domain. For now, ignore the security label on line 4, which uses a new syntax explained later in this section. Grouping variables in this way makes code clearer and more compact. Unfortunately, SecVerilog cannot precisely capture the desired label for the resulting packet. In this example, the address is public, but depending on the value of the `ns` bit, the data could be confidential. Thus, the (static) security levels of some of the bits in the packet depend on the run-time value of other bits.

The upper 11 bits of the packet have a different label than the rest of the packet, but SecVerilog applies the same label to all bits in a bit vector. Lowering the type of the entire packet is not a solution because the secrecy of the data must be protected when `id` indicates that the data is confidential. Similarly, raising the entire type to `secret` does not work because the address is used to make routing decisions that are observable to both worlds.

Our solution is to enrich what can be expressed using dependent labels, as shown in braces on line 4. The label expression specifies that the security class

of the i^{th} bit depends both on i and on the value of the `ns` bit. The type of `packet` is a function that takes an integer, i, representing an index to the bit vector. If the index is less than 31, the data is accessed, so it returns a type that depends on the MSB of the packet that corresponds to the `ns` bit. Otherwise, the address or the `ns` bit is accessed, so the returned type is `PT`.

Figure 2.5 shows the formal type syntax of SecVerilog2. Crucially, the type system of SecVerilog2 is extended with types of higher kinds. Kinds, written k, can either be levels $\ell \in \mathcal{L}$ or partial functions from integers to other kinds. In the SecVerilog syntax, all types, including dependent types (which are functions that are fully applied to variables) are of the kind ℓ.

Types (i.e., labels) in the extended syntax, written τ, are pure (side-effect-free) expressions signifying security levels. The syntax $v \mapsto \tau$ specifies a mapping from a position in the bit vector to the type of the bit at that position, and is used to specify the type of `packet` in Figure 2.4. Since $v \mapsto \tau$ is a form of function abstraction, types written with this syntax are higher-kinded. Dependent types may be written by referring to program variables in the type expression. The syntax `if` e^{τ} τ_t τ_f and `case` e^{τ} τ_1 ...τ_n describe conditional selection between security labels. The syntax of e^{τ} describes pure expressions and is omitted because it is standard. However, notably e^{τ} may contain variables declared in the program, and therefore, can be used to write dependent types.

At a high level, the type system is extended to track the bit-width of each variable in addition to its type. Types of kind ℓ are lifted in the obvious way to `int` $\rightharpoonup \ell$, so that all types become functions from bit indices to labels. During assignment checking, the bit-width of both sides of the assignment is used as a range for quantification.

$$\text{T-CONST}\ \frac{}{\Gamma;\mathtt{W};\Theta \vdash \mathtt{n} : \bot, \mathtt{w}} \qquad \text{T-VAR}\ \frac{\Gamma(x) = \tau \qquad \mathtt{W}(\mathtt{x}) = \mathtt{w}}{\Gamma;\mathtt{W};\Theta \vdash \mathtt{x} : \tau, \mathtt{w}}$$

$$\text{T-LOGICAL}\ \frac{\begin{array}{c}\Gamma;\mathtt{W};\Theta \vdash \mathtt{e}_1 : \tau_1, \mathtt{w} \qquad \Gamma;\mathtt{W};\Theta \vdash \mathtt{e}_2 : \tau_2, \mathtt{w} \\ \Theta \vdash \tau_1 : \mathtt{int} \rightharpoonup \ell \qquad \Theta \vdash \tau_2 : \mathtt{int} \rightharpoonup \ell \\ \tau = i \mapsto (\tau_1\ i) \sqcup (\tau_2\ i)\end{array}}{\Gamma;\mathtt{W};\Theta \vdash \mathtt{e}_1\ \mathtt{bop}\ \mathtt{e}_2 : \tau, \mathtt{w}}\ (\text{when } \mathtt{bop} \in \{\vee \wedge \oplus\})$$

$$\text{T-ARITH}\ \frac{\begin{array}{c}\Gamma;\mathtt{W};\Theta \vdash \mathtt{e}_1 : \tau_1, \mathtt{w} \qquad \Gamma;\mathtt{W};\Theta \vdash \mathtt{e}_2 : \tau_2, \mathtt{w} \\ \Theta \vdash \tau_1 : \mathtt{int} \rightharpoonup \ell \qquad \Theta \vdash \tau_2 : \mathtt{int} \rightharpoonup \ell \\ \tau = i \mapsto \bigsqcup_{j \in (1,i)} ((\tau_1\ j) \sqcup (\tau_2\ j))\end{array}}{\Gamma;\mathtt{W};\Theta \vdash \mathtt{e}_1\ \mathtt{bop}\ \mathtt{e}_2 : \tau, \mathtt{w}}\ (\text{when } \mathtt{bop} \in \{+ -\})$$

$$\text{T-CONCAT}\ \frac{\begin{array}{c}\Gamma;\mathtt{W};\Theta \vdash \mathtt{e}_1 : \tau_1, \mathtt{w}_1 \qquad \Gamma;\mathtt{W};\Theta \vdash \mathtt{e}_2 : \tau_2, \mathtt{w}_2 \\ \Theta \vdash \tau_1 : \mathtt{int} \rightharpoonup \ell \qquad \Theta \vdash \tau_2 : \mathtt{int} \rightharpoonup \ell \\ \tau = i \mapsto \mathtt{if}(i > w2)\ (\tau_1\ i - w_2 + 1)\ (\tau_2\ i)\end{array}}{\Gamma;\mathtt{W};\Theta \vdash \{\mathtt{e}_1; \mathtt{e}_2\} : \tau, (\mathtt{w}_1 + \mathtt{w}_2)}$$

$$\text{T-LSHIFT}\ \frac{\Gamma;\mathtt{W};\Theta \vdash \mathtt{e} : \tau, \mathtt{w} \qquad \Theta \vdash \tau : \mathtt{int} \rightharpoonup \ell \qquad \tau' = i \mapsto \mathtt{if}\ (i > n)\ (\tau\ i - n + 1)\ \bot}{\Gamma;\mathtt{W};\Theta \vdash \mathtt{e} << \mathtt{n} : \tau', \mathtt{w}} \qquad \text{T-RSHIFT}\ \frac{\Gamma;\mathtt{W};\Theta \vdash \mathtt{e} : \tau, \mathtt{w} \qquad \Theta \vdash \tau : \mathtt{int} \rightharpoonup \ell \qquad \tau = i \mapsto \mathtt{if}\ (i > w - n)\ \bot\ (\tau\ i + n)}{\Gamma;\mathtt{W};\Theta \vdash \mathtt{e} >> \mathtt{n} : \tau, \mathtt{w}}$$

$$\text{T-ARRINDEX}\ \frac{\begin{array}{c}\Gamma(x) = \tau_x \qquad \mathtt{W}(\mathtt{x}) = \mathtt{w} \\ \Gamma;\mathtt{W};\Theta \vdash \mathtt{e} : \tau_\mathtt{e}, \mathtt{w}_\mathtt{e} \\ \Theta \vdash \tau_x : \mathtt{int} \rightharpoonup \mathtt{int} \rightharpoonup \ell \\ \Theta \vdash \tau_e : \mathtt{int} \rightharpoonup \ell \\ \tau'_e = i \mapsto \bigsqcup_{i \in (1, w_e)} \tau_e\ i\end{array}}{\Gamma;\mathtt{W};\Theta \vdash \mathtt{x}[\mathtt{e}] : \tau'_\mathtt{e} \sqcup (\tau_\mathtt{x}\ \mathtt{e}), \mathtt{w}}$$

Figure 2.6: Typing rules for SecVerilog2 expressions.

The type rules of SecVerilog2 expressions are shown in Figure 2.6. In addition to a standard type environment Γ, a width environment $\mathtt{W}$ maps variables to their bit-widths. The width environment is populated with the declared widths of variables. Bit-widths are static, finite, and specified with integer constants. Since Verilog does not support dynamically-sized bit vectors, ranges are easily determined at compile time. Typing judgments for expressions have the form

$\Theta; \Gamma; \mathtt{W} \vdash \mathtt{e} : \tau, \mathtt{w}$ meaning that under context $\Theta; \Gamma; \mathtt{W}$, expression e has type τ and bit-width w. A kind environment, Θ, is used to make kind judgments of the form $\Theta \vdash \tau : k$.

The rule T-LOGICAL for logical binary operators checks that the widths of both expressions are the same and that both expressions have $\mathtt{int} \rightharpoonup \ell$ types. The type of the resulting expression is the bitwise join of the types of the operands. The rule T-ARITH must track the bits that are propagated by carry bits. The i^{th} bit of the result is affected by all bits below i from both inputs. The rule for concatenations (T-CONCAT) selects between the type functions of the original sub-expressions, shifting the upper expression as needed. The rules for shifting by constants (T-LSHIFT and T-RSHIFT) select the bottom type for the bits of the resulting expression that are constant. For the remaining parts, the type function of the non-constant sub-expression is shifted. The rule for indexed arrays (T-ARRINDEX) is discussed in Section 2.2.2.

Per-bit checking is done in the type-checking rule for assignments. The check verifies that the type of each bit of the right side of the assignment (joined with the program counter) is lower than the corresponding bit on the left. To type-check an assignment of some expression with type τ_r to a variable x, with type τ_l, both of these types are applied to each integer within $(0, \mathtt{W}(\mathtt{x}))$. If the condition $\forall i \in (0, \mathtt{W}(\mathtt{x})).\tau_{\mathtt{r}}(\mathtt{i}) \sqcup \mathtt{pc}(\mathtt{i}) \sqsubseteq \tau_{\mathtt{l}}(\mathtt{i})$ holds, the check succeeds. We omit the rules for commands since they are straightforward. Notably, pc is a bitwise label that is determined by commands in a straightforward way. Using a bitwise label for pc is more permissive than alternative rules which might compute the join over the bitwise labels of expressions, for example, used as conditionals in if-statements.

```
1  ...
2  reg {public} reg_id  [0:1023];
3  reg [0:31] { i -> j -> domain(reg_id[i]) } mem[0:1023];
4  ...
5
6  if(read_id == 1) begin
7      read = (reg_id[read_addr] == 1) ?
8          mem[read_addr] : 32'b0;
9  end else begin
10 ...
```

Figure 2.7: A cache code segment.

2.2.2 Per-element types

Arrays are commonly used in hardware descriptions. However, prior secure HDLs had minimal support for labeling arrays; all elements must have the same type. The code segment shown in Figure 2.7 describes part of the memory array of a cache that implements reads. The input `id` is the security id of the device originating the read request. The output `read` is the output data, which has a type that depends on `read_id`. This code is secure, but cannot be written in SecVerilog.

Following common practice, the cache is implemented as an array of memory cells, `mem`. Another array `reg_id` stores the id of the last device to write to each address of the array. Therefore, the label of a particular memory cell at array position i should depend on `reg_id`. With support for fine-grained array labels, the memory cells can be implemented conveniently as an array of bit vectors.

To support arrays in which each element has a distinct type, array variables must have kind $\texttt{int} \rightharpoonup \texttt{int} \rightharpoonup \ell$ when they are declared. In the rule for indexed arrays (T-ARR-INDEX), the type of the array, τ_x, is applied as a function to the expression that indexes the array, `e`. Doing so produces the security label of

the selected element of the array. The requirement imposed on the kind of τ_x ensures that $\tau_x\ e$ is $\texttt{int} \rightharpoonup \ell$, which is a mapping from the bit position to the label of that bit. Arrays in Verilog may only be indexed by variables and constants rather than arbitrary expressions, and therefore e can be substituted into τ_x at compile time. The label τ'_e is the (bitwise) join over the $\texttt{int} \rightharpoonup \ell$ label of `e`. Since the value of `e` determines which element of `x` is selected, each bit of `e` affects the value of `x[e]`. So τ'_e is used to elevate the label of `x[e]` to reflect that each bit of `e` has influenced `x[e]`

On line 3 of Figure 2.7, `mem` has an array type that maps the index of the array to a type that depends on the value stored in `reg_ns` at the corresponding index. Although the function does not depend on j, it is still written as a type function of kind $\texttt{int} \rightharpoonup \texttt{int} \rightharpoonup \ell$, so that when the array index is applied, the type can serve as a mapping from bits to types. To support arrays of bit vectors where each element has a different mapping from indices to types, the type function can be written to depend on both i and j.

2.2.3 Bundle labels

Wires in Chisel can also be grouped into a bundle. Bundles behave much like record types or structs; they consist of a set of field names which are mapped to data elements. Bundles are useful for describing packets or for grouping together the signals that correspond to just the subset of a port that accepts requests or just the subset that emits responses. As a result, it is naturally desirable to annotate different fields with different labels.

ChiselFlow supports heterogeneously labeled bundles as shown in Fig-

```
class DCacheDataReq(val lblParam: Label)(implicit p: Parameters)
    extends L1CacheBundle()(p) with ParamLabeled {
  val dconf = Bits(hcWidth.W, lblParam )
  val dinteg = Bits(hcWidth.W, lblParam )

  val wdata = Bits(rowBits.W, hlvl(dconf, dinteg) join lblParam )
  val addr = Bits(untagBits.W, lblParam )
  val write = Bool(lblParam )
  val wmask = Bits(rowBytes.W, lblParam )
  val way_en = Bits(nWays.W, lblParam)
  val take_dtag = Bool(lblParam )
  val sw_dwn = Bool(lblParam)

  override def cloneType = (new DCacheDataReq(lblParam)(p)).asInstanceOf[this.type]
  def cloneWithLabel(l: Label) = (new DCacheDataReq(l)(p)).asInstanceOf[this.type]
}

class DCacheDataArray(implicit p: Parameters) extends L1CacheModule()(p) {
  val io = IO(new Bundle with HCLabeledIn {
    val req = Input(ParamValid(new DCacheDataReq(lvl)))
    val resp = Output(Vec(nWays, new DCacheDataResp(lvl)))
  })
  //... the body of the data array is defined here
}
```

Figure 2.8: Heterogeneously labeled bundles in ChiselFlow.

ure 2.8. This figure shows a bundle that describes a data cache array request port and part of a data cache array that describes and instantiates its ports. This example is taken from the HyperFlow processor described in Chapter 4 . The `DCacheDataReq` class describes a bundle type because it extends from a descendent of the Chisel `Bundle` class. As in Chisel, the fields of the bundle are described by instantiating members of Chisel data classes such as `Bits` and `Bool`. Members of a bundle can be labeled by passing a Label object to the second argument of the data element.

The `DCacheDataReq` is heterogeneously labeled. Most members of the bundle are labeled with a parameter called `lblParam`. Here, `lblParam` represents the level of secrecy that can be observed by measuring the timing of events related to the request, such as the time that the request is valid. The field, `wdata`, however, has a different label because it includes write-back data from either

the processor pipeline or the memory hierarchy. The inputs `dconf` and `dinteg` represent dynamic confidentiality and integrity levels for `wdata`, and the label of `wdata` is the join of `lblParam` and a function `hlvl` that maps the dynamic levels to a label.

The class `DCacheDataArray` is a module that describes the data array for the data cache. Its port, `io`, is also a bundle because it is an anonymous class that extends from Bundle. The IO port includes a `DCacheDataReq` input wrapped in a valid interface. The `ParamValid` class creates a new bundle that adds a `valid` signal to its argument. The `valid` signal is asserted whenever the argument represents useful data that can be consumed. The IO port also includes a vector of `nWays` data cache responses. The type of the responses is described by a class called `DCacheDataResp`, which is not shown.

By extending from the trait, `HCLabeledIn`, the IO port also includes signals that represent a dynamic timing levels, and a label called `lvl` that maps these signals to a label. The requests and responses of the data cache array take label parameters as arguments. In the IO port, `lvl` is passed as the label parameter for both so that that request input and all of the outputs have the same timing label. The request and response classes both describe types in which all members are labeled. The signals described by the `HCLabeledIn` trait are also labeled. As a result, the IO port describes a well-formed type in which all members of the port are fully labeled. The port is also labeled heterogeneously because the data inputs and outputs have labels that differ from the other members.

Fields of a bundle such as `DCacheDataArray` can be referenced by simply referencing the member of the class that represents that field. For example, if `cacheReq` has type `DCacheDataReq`, then `cacheReq.wdata` refers to the data ele-

ment of `cacheReq`. Type declarations, such as the type of `IO` are emitted by ChiselFlow as SIRRTL code and include the labels for each element of the `IO` port. The types and labels of expressions, such as `cacheReq.wdata` are determined by SIRRTL. Chisel also supports parallel assignments, in which one bundle can be assigned from another bundle of the same type. For example,

$$\texttt{req\_pipe1} := \texttt{cacheReq},$$

might pipeline an entire cache request. Parallel assignments such as this one are type-checked by un-rolling the parallel assignment into distinct assignments of each record.

2.3 Non-malleable downgrading

Noninterference [33] is too restrictive for practical systems. Values computed using secrets need to eventually be released to the public – for example, an encrypted value can be safely released even though it depends on the encryption key. Similarly, untrusted values need to be permitted to influence trusted state, for example, after they have been sanitized. As a result, all practical systems for information flow control permit *downgrades* which relax information flow policies. Downgrading that relaxes confidentiality is called *declassification*, whereas downgrading that relaxes integrity is called *endorsement* [103].

Because downgrades weaken noninterference, effort has been made to constrain downgrading to limit its potential to cause harm [75]. SIRRTL constrains declassification to enforce robust declassification [101]. Roughly, robust declas-

sification prevents a low-integrity attacker from influencing what information is declassified. In label models that incorporate principals [15, 10], more generally restricts a party to declassifying information that it has sufficient privilege or authority to write. Dually, we constrain endorsements so that they are transparent [10]. Endorsement is transparent if it does not allow secret data to be endorsed in a public context. More generally, a principal is permitted to endorse data that it has sufficient authority or privilege to read. Both forms of constrained downgrading have been shown to prevent attacks [10]. The joint enforcement of robust declassification and transparent endorsement is known as non-malleable information flow control [10].

As in prior work on defining robust declassification, authority or privilege to release information in SIRRTL is represented by integrity [15, 10]. Naturally, the ability to read, and therefore transparently endorse, a piece of information is represented by confidentiality [10]. Therefore, the labels of SIRRTL are product labels that include confidentiality and integrity levels.

SIRRTL levels, ℓ, are therefore pairs of confidentiality and integrity components (c, i). Label components include atomic security levels n which form a lattice. The notation $p_1 \succeq p_2$ means that p_1 has higher authority than p_2. For confidentiality, $c_1 \succeq c_2$ means that c_1 is more secret than c_2. Dually, for integrity, $i_1 \succeq i_2$ means that i_2 is more trusted than i_2. For either confidentiality or integrity components, $p_1 \wedge p_2$ denotes the join of p_1 and p_2, and $p_1 \vee p_2$ denotes their meet. The greatest and least components in $\succeq$ order are $\top$ and $\bot$ respectively. The lattice over label components is lifted to a lattice over security labels that is defined in Figure 2.9

The syntax `decl`(e, ℓ) and `endo`(e, ℓ) respectively express declassification and

$$(c_1, i_1) \sqcup (c_2, i_2) \triangleq (c_1 \wedge c_2, i_1 \vee i_2)$$
$$(c_1, i_1) \sqcap (c_2, i_2) \triangleq (c_1 \vee c_2, i_1 \wedge i_2)$$
$$(c_1, i_1) \sqsubseteq (c_2, i_2) \iff c_2 \succeq c_1 \text{ and } i_1 \succeq i_2$$

Figure 2.9: Lattice over labels.

endorsement of the expression e to the security level ℓ. The typing rules for downgrades enforce non-malleable information flow control and are similar to those used to enforce the same security condition in a recent functional programming language [10]. These rules require auxiliary definitions on labels. In particular, $\ell^{\rightarrow}$ is defined $(c, i)^{\rightarrow} \triangleq (c, \top)$, and it computes a label that has the confidentiality of ℓ, but is fully trusted. Similarly, $\ell^{\leftarrow}$ is defined $(c, i)^{\leftarrow} \triangleq (\bot, i)$ and it computes a label that has the integrity of ℓ but is fully public. The view of a label, $\Delta(\ell)$ converts the integrity of a label to a confidentiality component, and is defined by $\Delta(c, i) \triangleq (i, \top)$. Dually, the voice of a label, $\nabla(\ell)$ converts a confidentiality component to an integrity component, and it is defined by $\nabla(c, i) \triangleq (\bot, c)$.

To declassify an expression from label ℓ' to label ℓ, the type system ensures that

$$\ell'^{\rightarrow} \sqsubseteq \ell^{\rightarrow} \sqcup \Delta(\ell' \sqcup pc)$$

This condition follows directly from prior work on defining robust declassification in the context of programming languages [15, 10]. Roughly, it allows the confidentiality $\ell^{\rightarrow}$ of the data being declassified to be "made up for" by the integrity $\ell^{\leftarrow}$ of the data being declassified and the integrity $\ell^{\leftarrow}_{cur}$ of the current process.

Dually, to transparently endorse an expression from label ℓ' to label ℓ, the

type system checks that

$$\ell'^{\leftarrow} \sqsubseteq \ell^{\leftarrow} \sqcup \nabla(\ell' \sqcup pc)$$

This condition sets a maximum confidentiality on endorsements to prevent opaque writes that could enable attacks. A write is opaque if a principal could have written data but not read it.

2.4 Formalism and security results for SIRRTL

We now define a syntax, semantics, and type rules for a core subset of SIRRTL in order to establish security results about well-typed SIRRTL modules. Because SIRRTL is responsible for enforcing the security policies described by ChiselFlow, the same security results are enjoyed by ChiselFlow code. This core subset is also sufficient to capture the same novel features in SecVerilog2, and as a result, the security results also hold for the implementation of SecVerilog2. In particular, we prove that well-typed SIRRTL modules that do not contain downgrades enforce a timing-sensitive variant of observational determinism.

2.4.1 Syntax

Figure 2.10 shows a core syntax of SIRRTL that we use to establish security results. The syntax of labels has already been described. Aside from atomic security levels, the syntax of label components also includes functions f that are fully applied to some number of free variables $\vec{x}$ in the hardware module. Components of this form can be used to express dependent labels that change at run time based on the values of signals. Dependent labels are important for the

description of efficient hardware designs that allow the hardware to be shared by different security domains at run-time. Dependent labels of this form are similar to those in SecVerilog [110].

$$
\begin{array}{rr}
n \in \mathcal{N} & \text{atomic principals} \\
x \in \mathcal{V} & \text{variable names} \\
\bar{x} \in \overline{\mathcal{V}} & \text{next-cycle symbols} \\
v \in \mathbb{N} & \text{integers}
\end{array}
$$

$$
\begin{array}{rl}
i, c, p ::= & n \mid \top \mid \bot \mid p \wedge p \mid p \vee p \mid f(\vec{x}) \\
\ell ::= & (p, p) \\
e ::= & v \mid x \mid \bar{x} \mid e \oplus e \mid \texttt{decl}(e, \ell) \mid \texttt{endo}(e, \ell) \\
\texttt{Prog}, s ::= & \texttt{skip} \mid s; s \mid \texttt{when}(e)\ s\ \texttt{else}\ s \mid x \Leftarrow e
\end{array}
$$

Figure 2.10: SIRRTL core syntax.

The syntax of expressions is mostly standard. Values v are finite bit-vectors. Variables, denoted x, represent sequential variables that define registers. For simplicity, the formal syntax of SIRRTL omits combinational variables, though in doing so it loses no expressive power – combinational variables can simply be replaced with the expressions that define their values. The implementation of SIRRTL includes combinational variables. The syntax $\bar{x}$ represents a special symbol reserved for storing the next-cycle valuation of x. The symbol $\bar{x}$ cannot be written by the programmer; it is an auxiliary symbol used in the semantics and typing judgments to capture delayed updates to the sequential variables. Binary operators are denoted $e_1 \oplus e_2$. As has already been described, the syntax $\texttt{decl}(e, \ell)$ and $\texttt{endo}(e, \ell)$ respectively express declassification and endorsement of the expression e to the security level ℓ The syntax of statements s is entirely

standard except that `when` denotes a conditional statement. Programs, written `Prog` are single commands.

Our full implementation of SIRRTL (and ChiselFlow) also securely supports record types (bundles), arrays, module declarations and instantiations, and purely combinational wires, though the typing rules for these features do not fundamentally change the design of the type system, so we omit them from the core syntax. The heterogeneous bit vector and array labels described in Section 2.2.1 and Section 2.2.2 are interesting features because they represent dependent labels with function bindings. We prove that the core syntax extended with support for bit vector and array labels enjoys the same security guarantees as SIRRTL by showing that there is a label-preserving and clearly semantics preserving translation from the extended language to the core language.

2.4.2 Semantics

The big-step semantics of expressions, shown in Figure 2.10 is entirely standard aside from the rule for $\bar{x}$ which is similar to the evaluation of a conventional variable. The small-step semantics of statements, shown in Figure 2.12 is mostly standard. Hardware states, σ, are mappings from variables and next-cycle symbols to bit-vectors. Formally, states range over $(\mathcal{V} + \overline{\mathcal{V}}) \to \mathbb{N}$, or isomorphically, $(\mathcal{V} \to \mathbb{N}) \times (\overline{\mathcal{V}} \to \mathbb{N})$; they are are pairs including a function from variables to values and a function from next-cycle symbols to values. For simplicity, we use $\sigma(x)$ and $\sigma(\bar{x})$ to denote the valuation of either a variable or next-cycle symbol in σ. Assignments to a variable x cause updates to $\bar{x}$ rather than x to model the fact that updates to registers are delayed until the start of the clock cycle.

Variables are updated by the program semantics. As the program is evaluated, the program semantics constructs traces that have the syntax

$$t ::= \epsilon \mid (T, \sigma) \mid t_1; t_2$$

where T is a clock cycle counter represented by a positive integer. The program semantics operates on configurations of the form $\langle T, \sigma, c, t \rangle$. Transitions between configurations are denoted by $\rightarrow_{\mathcal{S}}$, in which $\mathcal{S}$ represents the statement that is the initial syntactic description of the program.

The rule, $S - Tick$ applies when the program has been fully evaluated to `skip`. This rule updates the contents of the registers on the new clock cycle. All variables $x_1, ..., x_n$ in the program $\mathcal{S}$, are updated to their corresponding next-cycle valuations stored in $\sigma x_1, ..., \sigma x_n$. The cycle counter is incremented, and a trace event that includes the clock cycle number and the state at the start of the cycle is emitted. The program re-starts evaluation from $\mathcal{S}$ to compute the values for the next cycle. The rule, $S - Eval$, applies when s is not `skip`, and it simply updates the statement and state according to the semantics of statements.

$$\frac{}{\langle \sigma, n \rangle \Downarrow n}\ \text{S-CONST} \qquad \frac{\sigma(x) = n}{\langle \sigma, x \rangle \Downarrow n}\ \text{S-VAR} \qquad \frac{\sigma(\bar{x}) = n}{\langle \sigma, \bar{x} \rangle \Downarrow n}\ \text{S-VARNEXT}$$

$$\frac{\langle \sigma, e_1 \rangle \Downarrow n_1 \qquad \langle \sigma, e_2 \rangle \Downarrow n_2 \qquad n = n_1 \oplus n_2}{\langle \sigma, e_1 \oplus e_2 \rangle \Downarrow n}\ \text{S-OP} \qquad \frac{\langle \sigma, e \rangle \Downarrow n}{\langle \sigma, \texttt{decl}(e, \ell) \rangle \Downarrow n}\ \text{S-DECL}$$

$$\frac{\langle \sigma, e \rangle \Downarrow n}{\langle \sigma, \texttt{endo}(e, \ell) \rangle \Downarrow n}\ \text{S-ENDO}$$

Figure 2.11: Expression semantics.

$$
\begin{aligned}
\langle \sigma, x \Leftarrow e \rangle &\longrightarrow \langle \sigma[\bar{x} \mapsto \sigma(x)], \texttt{skip} \rangle \\
\langle \sigma, \texttt{skip}; s \rangle &\longrightarrow \langle \sigma, s \rangle \\
\langle \sigma, s_1; s_2 \rangle &\longrightarrow \langle \sigma', s_1'; s_2 \rangle && (\text{if } \langle \sigma, s_1 \rangle \longrightarrow \langle \sigma', s_1' \rangle) \\
\langle \sigma, \texttt{when}(e)\ s_1\ \texttt{else}\ s_2 \rangle &\longrightarrow \langle \sigma, s_1 \rangle && (\text{if } \neg(\langle \sigma, e \rangle \Downarrow 0)) \\
\langle \sigma, \texttt{when}(e)\ s_1\ \texttt{else}\ s_2 \rangle &\longrightarrow \langle \sigma, s_2 \rangle && (\text{if } \langle \sigma, e \rangle \Downarrow 0)
\end{aligned}
$$

Figure 2.12: Command semantics.

$$
\frac{\begin{array}{c}\{x_1, ..., x_n\} = vars(\mathcal{S}) \\ \sigma' = \sigma[x_1 \mapsto \sigma(\bar{x_1})]...[x_n \mapsto \sigma(\bar{x_n})]\end{array}}{\langle T, \sigma, \texttt{skip}, t \rangle \rightarrow_{\mathcal{S}} \langle T+1, \sigma', \mathcal{S}, t; (T+1, \sigma') \rangle}\ \text{S-TICK}
$$

$$
\frac{s \neq \texttt{skip} \qquad \langle \sigma, s \rangle \rightarrow \langle \sigma', s' \rangle}{\langle T, \sigma, s, t \rangle \rightarrow_{\mathcal{S}} \langle T, \sigma', s', t \rangle}\ \text{S-EVAL}
$$

Figure 2.13: Program semantics.

2.4.3 Type rules

Type environments Γ map variables and next-cycle symbols to labels. Because label components in SIRRTL include functions of program variables, the valuation of components and labels both depend on the state, σ. We use the meta-syntax $\mathcal{C}(p, \sigma)$ and $\mathcal{T}(\ell, \sigma)$ respectively to denote the valuations of components p and labels ℓ respectively. As in SecVerilog[110], the type rules apply only for type environments that are well-formed. In a well-formed type environment 1) no label depends on variables with more restrictive labels, and 2) variables that appear in labels cannot depend on labels. The second condition is more restrictive than the one in SecVerilog, which allows variables to have labels that depend on themselves. Let $fv(\ell)$ denote the free variables in ℓ. Formally, a type-environment is well-formed, written $\vdash \Gamma$ when,

Definition 1 (Well-Formedness of Environments)

$$\forall x \in \mathcal{V}.(\forall \sigma.\forall x' \in fv(\Gamma(x)).$$
$$\mathcal{T}(\Gamma(x'), \sigma) \sqsubseteq \mathcal{T}(\Gamma(x), \sigma)$$
$$\wedge\, fv(\Gamma(x')) = \emptyset)$$

Type judgements for expressions have the form $\Gamma; pc \vdash e : \ell$ which means that e is well-typed in typing environment Γ under program counter label pc. The type rules for expressions are mostly standard aside from next-cycle valuations of variables and for downgrades. The rule $T - NextVar$ computes the valuation of the label x on the following clock cycle by substituting each occurrence of a free variable in the label with its next-cycle symbol.

Because labels in SIRRTL can depend on the run-time values of signals, SIRRTL relies on a static program analysis that that models the run-time behavior of the hardware. The program analysis uses a predicate transformer semantics to conservatively approximate the strongest postcondition at each point in the module. The notation $P(\eta) \Rightarrow Q$ means that the program analysis has derived that the proposition Q holds before executing control-flow graph node η.

The rules for downgrades enforce non-malleable information flow control and have mostly been described in Section 2.3. In addition to the premises that have already been described in Section 2.3, the type rules for declassification and endorsement also require $pc \sqsubseteq \ell$ to ensure that the downgrades is not influenced by low-integrity values indirectly through control flow, and that implicit flows are prevented.

The typing rules for commands are standard except that the rule for assignments which differs subtly from a standard typing judgement for assign-

ment. Assignments to sequential variables describe the values which they store at the start of the next clock edge. Therefore, the label of the expression ℓ should be permitted to flow into the label of the sequential variable during the next clock cycle ℓ'. This new label may depend on the new values of other sequential variables. Since the actual values of variables are not known a priori, ℓ' is determined by simultaneously substituting each sequential variable x_i in $\Gamma(x)$ with its corresponding next-cycle value symbol $\bar{x_i}$. Thus, $\ell' = \Gamma(x)[x_1 \mapsto \bar{x_1}]...[x_n \mapsto \bar{x_n}]$ and $P(\eta)$ must contain sufficient facts to fulfill the proof obligation that this flow is safe.

$$\text{T-CONST}\ \frac{}{\Gamma; pc \vdash n : \bot} \qquad \text{T-VAR}\ \frac{\Gamma(x) = \ell}{\Gamma; pc \vdash x : \ell}$$

$$\text{T-NEXTVAR}\ \frac{\Gamma(x) = \ell \qquad \{x_1, ..., x_n\} = fv(\Gamma(x))}{\Gamma; pc \vdash \bar{x} : \ell[x_1 \mapsto \bar{x_1}]...[x_n \mapsto \bar{x_n}]} \qquad \text{T-OP}\ \frac{\Gamma; pc \vdash e_1 : \ell_1 \quad \Gamma; pc \vdash e_2 : \ell_2}{\Gamma; pc \vdash e_1 \ \oplus \ e_2 : \ell_1 \sqcup \ell_2}$$

$$\text{T-DECL}\ \frac{\Gamma; pc \vdash e : \ell' \qquad P(\eta) \Rightarrow \ell^{\leftarrow} = \ell'^{\leftarrow} \wedge pc \sqsubseteq \ell \quad P(\eta) \Rightarrow \ell'^{\rightarrow} \sqsubseteq \ell^{\rightarrow} \sqcup \Delta(\ell' \sqcup pc)}{\Gamma; pc \vdash \texttt{decl}(e, l) : l}$$

$$\text{T-ENDO}\ \frac{\Gamma; pc \vdash e : \ell' \qquad P(\eta) \Rightarrow \ell^{\rightarrow} = \ell'^{\rightarrow} \wedge pc \sqsubseteq \ell \quad P(\eta) \Rightarrow \ell'^{\leftarrow} \sqsubseteq \ell^{\leftarrow} \sqcup \nabla(\ell' \sqcup pc)}{\Gamma; pc \vdash \texttt{endo}(e, l) : l}$$

Figure 2.14: Type Rules: Expressions.

2.5 Security results for core SIRRTL language

We now prove security results about SIRRTL, namely that well-typed hardware modules that do not contain downgrades enforce a timing-safe variant of ob-

$$\frac{}{\Gamma; pc \vdash \texttt{skip}}\ \text{T-SKIP} \qquad \frac{\Gamma; pc \vdash s_1 \qquad \Gamma; pc \vdash s_2}{\Gamma; pc \vdash s_1; s_2}\ \text{SEQ}$$

$$\frac{\begin{array}{c}\Gamma; pc \vdash e : \ell \\ \Gamma; pc \sqcup \ell \vdash s_t \\ \Gamma; pc \sqcup \ell \vdash s_f\end{array}}{\Gamma; pc \vdash \texttt{when}(e)\ s_t\ \texttt{else}\ s_f}\ \text{T-WHEN}$$

$$\frac{\begin{array}{c}\Gamma; pc \vdash e : \ell \qquad \{x_1, ..., x_n\} = fv(\Gamma(x)) \\ \ell' = \Gamma(x)[x_1 \mapsto \bar{x_1}]...[x_n \mapsto \bar{x_n}] \\ P(\eta) \Rightarrow \ell \sqcup pc \sqsubset \ell'\end{array}}{\Gamma; pc \vdash x \Leftarrow e}\ \text{T-ASSIGN}$$

Figure 2.15: Type Rules: Statements.

servational determinism. The typing rules for downgrades in SIRRTL also resemble those from a recent software type system that enforces a security condition in the presence of downgrades called non-malleable information flow control [10]. We first define low-equivalence of hardware states before stating the main theorems. We define a full-evaluated security label as one which does not contain sub-components of the form $f(x)$. When two states, σ_1 and σ_2 are low-equivalent to an attacker at fully-evaluated security label L we write $\sigma_1 \approx_L \sigma_2$. Low-equivalence at level L is defined as follows,

$$\sigma_1 \approx_L \sigma_2 \triangleq \forall x \in \mathcal{V}.(\mathcal{T}(\Gamma(x), \sigma_1) \sqsubseteq L \iff \mathcal{T}(\Gamma(x), \sigma_2) \sqsubseteq L)$$
$$\wedge \mathcal{T}(\Gamma(x), \sigma_1) \sqsubseteq L \Longrightarrow \sigma_1(x) = \sigma_2(x)$$

Traces are low-equivalent, written $t_1 \approx_L t_2$ when for each element of the trace, the corresponding clock cycle counters are equal, and the states are low-equivalent.

We now state the observational determinism theorem.

Theorem 1 (Observational Determinism) *If Γ is a type environment, s is a statement that does not contain downgrades, pc is a label, L is a fully-evaluated security label, and σ_1 and σ_2 are states, then*

$$\begin{aligned}\vdash \Gamma \wedge \Gamma; pc \vdash s \wedge \sigma_1 \approx_L \sigma_2 \wedge & \\ \langle 0, \sigma_1, s, \epsilon\rangle \longrightarrow_{\mathcal{S}} \langle n_1, \sigma_1', s, t_1\rangle \wedge & \\ \langle 0, \sigma_2, s, \epsilon\rangle \longrightarrow_{\mathcal{S}} \langle n_2, \sigma_2', s, t_2\rangle & \\ \Longrightarrow \sigma_1' \approx_L \sigma_2' \wedge t_1 \approx_L t_2 &\end{aligned}$$

Before proving the observational determinism result, we first prove some useful lemmas. The first lemma states that low expressions other than downgrades do not contain high variables.

Lemma 1 *For all fully-evaluated security labels L, states σ, and expressions e that do not contain downgrades,*

$$\begin{aligned}\vdash \Gamma \wedge \Gamma \vdash e : \ell \wedge \mathcal{T}(\ell, \sigma) \sqsubseteq L & \\ \Longrightarrow \forall x \in vars(e). \mathcal{T}(\Gamma(x), \sigma) \sqsubseteq L &\end{aligned}$$

Proof. By induction on the structure of expressions. □

The next lemma states that low labels evaluate to the same concrete label in low-equivalent states.

Lemma 2 *If Γ is a type environment, ℓ is a label, L is a fully-evaluated label, and σ_1 and σ_2 are states, then*

$$\begin{aligned}\sigma_1 \approx_L \sigma_2 \wedge \vdash \Gamma \wedge \mathcal{T}(\ell, \sigma_1) \sqsubseteq L & \\ \Longrightarrow \mathcal{T}(\ell, \sigma_1) = \mathcal{T}(\ell, \sigma_2) &\end{aligned}$$

Proof. Let $\ell = (c, i)$ and $L = (c', i')$. By the definition of $\sqsubseteq$, $c' \succeq \mathcal{C}(c, \sigma_1)$ and $\mathcal{C}(i, \sigma_1) \succeq i'$. We show that $\mathcal{C}(c, \sigma_1) = \mathcal{C}(c, \sigma_2)$ by induction on the structure of c. The argument that $\mathcal{C}(i, \sigma_1) = \mathcal{C}(i, \sigma_2)$ is exactly dual, and the result that $\mathcal{T}(\ell, \sigma_1) = \mathcal{T}(\ell, \sigma_2)$ follows directly.

Case $c = n, c = \top, c = \bot$*:* trivial.

Case $c = f(\vec{x})$*:* Let x_i be some variable in $\vec{x}$. By assumption, $\sigma_1 \approx_{(c',i')} \sigma_2$. By $\vdash \Gamma$, $\mathcal{T}(\Gamma x_i, \sigma_1) \sqsubseteq \ell$ and $\mathcal{T}(\Gamma x_i, \sigma_2) \sqsubseteq \ell$ By transitivity of $\sqsubseteq$, $\mathcal{T}(\Gamma(x_i), \sigma_1) \sqsubseteq L$ and $\mathcal{T}(\Gamma(x_i), \sigma_2) \sqsubseteq L$. By definition of $\approx_L$, $\sigma_1(x_i) = \sigma_2(x_i)$. The same is true for all other variables in $\vec{x}$, and so $\mathcal{C}(f(\vec{x}), \sigma_1) = \mathcal{C}(f(\vec{x}), \sigma_2)$

Case $p_1 \wedge p_2$*:* $\mathcal{T}(p_1 \wedge p_2, \sigma_1) = \mathcal{T}(p_1, \sigma_1) \wedge \mathcal{T}(p_2, \sigma_2)$. By assumption $\mathcal{T}(p_1 \wedge p_2, \sigma_1) \succeq c'$, hence $\mathcal{T}(p_1, \sigma_1) \succeq c'$ and $\mathcal{T}(p_2, \sigma_2) \succeq c'$. By induction hypothesis, $\mathcal{T}(p_1, \sigma_1) = \mathcal{T}(p_1, \sigma_2)$ and $\mathcal{T}(p_1, \sigma_1) = \mathcal{T}(p_1, \sigma_2)$. Hence, $\mathcal{T}(p_1 \wedge p_2, \sigma_1) = \mathcal{T}(p_1 \wedge p_2, \sigma_2)$

Case $p_1 \vee p_2$*:* Similar to the case $f(\vec{x})$, by inspection of the free variables in $p_1 \vee p_2$. □

The next lemma states that low expressions evaluate to the same value in low-equivalent states.

Lemma 3 *If* Γ *is a type environment,* e *is an expression,* pc *is a label,* L *is a fully-*

evaluated security label, and σ_1 and σ_2 are states, then

$$\sigma_1 \approx_L \sigma_2 \wedge \vdash \Gamma \wedge \Gamma \vdash e : \ell \wedge \mathcal{T}(\ell, \sigma_1, \sqsubseteq) L \wedge \langle \sigma_1, e \rangle \Downarrow n_1 \wedge \langle \sigma_2, e \rangle \Downarrow n_2 \implies n_1 = n_2$$

Proof. By Lemma 2, $\mathcal{T}(\ell, \sigma_2) = \mathcal{T}(\ell, \sigma_1) \sqsubseteq L$. By Lemma 1, for all x in e, $\mathcal{T}(x, \sigma_1) \sqsubseteq L$ and $\mathcal{T}(x, \sigma_2) \sqsubseteq L$. Since $\sigma_1 \approx_L \sigma_2$, $\sigma_1(x) = \sigma_2 x$. Since this is true for all x in e, $n_1 = n_2$. □

We now prove that SIRRTL enforces observational determinism for individual statements.

Theorem 2 (Single-Statement Obsevational Determinism) *If Γ is a type environment, s is a statement that does not contain downgrades, pc is a label, L is a fully-evaluated security label, and σ_1 and σ_2 are states, then*

$$\vdash \Gamma \wedge \Gamma; pc \vdash s \wedge \sigma_1 \approx_L \sigma_2 \wedge \langle \sigma_1, s \rangle \longrightarrow_* \langle \sigma_1', skip \rangle \wedge \langle \sigma_2, s \rangle \longrightarrow_* \langle \sigma_2', skip \rangle \implies \sigma_1' \approx_L \sigma_2'$$

Proof. *Case $s_1; s_2$:* If $s_1 =$`skip`, then $\langle \sigma_1, s_1; s_2 \rangle \rightarrow \langle \sigma_1', s_2 \rangle$ and $\langle \sigma_2, s_1; s_2 \rangle \rightarrow \langle \sigma_2', s_2 \rangle$ so $\sigma_1 = \sigma_1'$ and $\sigma_2 = \sigma_2'$. By assumption, $\sigma_1 \approx_L \sigma_2$ and so $\sigma_1' \approx_L \sigma_2'$. By the induction hypothesis, execution of s_2 from σ_1' and σ_2' results in low-equivalent states.

If $s_1 \neq$`skip`, then $\langle \sigma_1, s_1; s_2 \rangle \rightarrow \langle \sigma_1'', s1'; s_2 \rangle$ and $\langle \sigma_2, s_1; s_2 \rangle \rightarrow \langle \sigma_2'', s1''; s_2 \rangle$ where $\langle \sigma_1, s_1 \rangle \rightarrow \langle \sigma_1'', s1' \rangle$ and $\langle \sigma_2, s_1 \rangle \rightarrow \langle \sigma_2'', s1'' \rangle$. By the induction hypothesis $\sigma_1'' \approx_L \sigma_2''$ But SIRRTL statements clearly do not diverge, so for some σ_1''' and

σ_2''', $\langle \sigma_1', s'1 \rightarrow_* \langle \sigma_1''', \texttt{skip} \rangle$ and $\langle \sigma_2', s'1 \rightarrow_* \langle \sigma_2''', \texttt{skip} \rangle$. By the induction hypothesis $\sigma_1''' \approx_L \sigma_2'''$. And since s_2 is eventually evaluated from σ_1''' and σ_2''' in two executions, by the induction hypothesis, $\sigma_1' \approx_L \sigma_2'$

Case $x \Leftarrow e$*:* We have $\langle \sigma_1, x \Leftarrow e \rangle \rightarrow \langle \sigma_1[\bar{x} \mapsto n_1], \texttt{skip} \rangle$ and $\langle \sigma_2, x \Leftarrow e \rangle \rightarrow \langle \sigma_2[\bar{x} \mapsto n_2], \texttt{skip} \rangle$ Let $\ell' = ell[x_1 \mapsto \bar{x_1}]...[x_n \mapsto \bar{x_n}]$. We first consider the case in which $\mathcal{T}(\ell', \sigma_1) \sqsubseteq L$. By assumption, $\sigma_1 \approx_L \sigma_2$ and by Lemma 2, $\mathcal{T}(\ell', \sigma_2) = \mathcal{T}(\ell', \sigma_1) \sqsubseteq L$ By T-ASSIGN, $\Gamma; pc \vdash e : \ell$ and $\ell \sqcup pc \sqsubseteq \ell'$. By lattice properties, $\ell \sqsubseteq \ell'$ and $\ell \sqsubseteq L$. By Lemma 3, $n_1 = n_2$. Because $\sigma_1[\bar{x} \mapsto n_1] = \sigma_1'$ and σ_1 agree on values of all variables other than $\bar{x}$, $\sigma_1 \approx_L \sigma_1'$. Similarly, $\sigma_2 \approx_L \sigma_2'$ and by transitivity, $\sigma_2' \approx_L \sigma_1'$.

We now consider the case in which $\mathcal{T}(\ell', \sigma_1) \not\sqsubseteq L$ We first show that $\mathcal{T}(\ell', \sigma_2) \not\sqsubseteq L$ If $\mathcal{T}(\ell', \sigma_2) \sqsubseteq L$, then by Lemma 2, $\mathcal{T}(\ell', \sigma_1) \sqsubseteq L$ which violates our assumption. By $\vdash \Gamma$, $\bar{x} \notin fv(Ga(\bar{x}))$. Because σ_1 and $\sigma_1[\bar{x} \mapsto n_1] = \sigma_1'$ agree on valuations of all variables other than $\bar{x}$, $\mathcal{T}(\ell', \sigma_1') \not\sqsubseteq L$. Similarly, $\mathcal{T}(\ell', \sigma_2') \not\sqsubseteq L$. Hence, $\sigma_1 \approx_l \sigma_1'$, and $\sigma_2 \approx_l \sigma_2'$, and by transitivity, $\sigma_1' \approx_l \sigma_2'$.

Case $\texttt{when}(e) s_1 else s_2$*:* By T-COND, $\Gamma; pc \vdash e : \ell$. We first consider the case in which $\mathcal{T}(\ell, \sigma_1) \sqsubseteq L$. By Lemma 2, $\mathcal{T}(\ell, \sigma_2) = \mathcal{T}(\ell, \sigma_1) \sqsubseteq L$. By Lemma 3, $\langle \sigma_1, e \rangle \Downarrow n$ and $\langle \sigma_2, e \rangle \Downarrow n$ for some n. By the semantics, either $\langle \sigma_1, s \rangle \rightarrow \langle \sigma_1, s1 \rangle$ and $\langle \sigma_2, s \rangle \rightarrow \langle \sigma_2, s1 \rangle$ or $\langle \sigma_1, s \rangle \rightarrow \langle \sigma_1, s2 \rangle$ and $\langle \sigma_2, s \rangle \rightarrow \langle \sigma_2, s2 \rangle$, but both executions take the same path. If the branch is taken, then by the induction hypothesis, $\langle \sigma_1, s \rangle \rightarrow_* \langle \sigma_1', \texttt{skip} \rangle$ and $\langle \sigma_2, s \rangle \rightarrow_* \langle \sigma_2', \texttt{skip} \rangle$ for some σ_1', σ_2' such that $\sigma_1' \approx_L \sigma_2'$. It is similar if the branch is not taken.

We now consider the case in which $\mathcal{T}(\ell, \sigma_1) \not\sqsubseteq L$. By Lemma 7, $\mathcal{T}(\ell, \sigma_2) \not\sqsubseteq L$. By T-COND, $\Gamma; pc \vdash s_1$. Let $pc' = pc \sqcup \ell$. Then $pc' \not\sqsubseteq L$ by lattice properties. Let x be some variable assigned in s_1. By T-ASSGN, $pc' \sqcup \ell'$ where $\ell' \neq \Gamma(x)[x_1 \mapsto \bar{x_1}]...[x_n \mapsto \bar{x_n}]$. By T-NEXTVAR, $\Gamma(\bar{x}) = \ell'$, and so $pc' \sqsubseteq \Gamma(\bar{x})$, and $Ga(\bar{x}) \not\sqsubseteq L$. The same is true for all other variables assigned in s_1 and for all variables assigned in s_2. Let $\langle \sigma_1, s_1 \rangle \rightarrow_* \langle \sigma_1'', \texttt{skip} \rangle$. Because only high variables are assigned in s_1, σ_1 and σ_1'' may only disagree on high variables, and so $\sigma_1'' \approx_L \sigma_1$. Similarly, $\sigma_2'' \approx_L \sigma_2$. Because $\sigma_1 \approx_l \sigma_2$, by transitivity twice, $\sigma_1'' \approx_L \sigma_2''$.

□

We now prove that well-typed SIRRTL modules enforce a timing safe variant of observational determinism

Theorem 1 (Observational Determinism) *If Γ is a type environment, s is a statement that does not contain downgrades, pc is a label, L is a fully-evaluated security label, and σ_1 and σ_2 are states, then*

$$\begin{aligned} \vdash \Gamma \wedge \Gamma; pc \vdash s \wedge \sigma_1 \approx_L \sigma_2 \wedge & \\ & \langle 0, \sigma_1, s, \epsilon \rangle \longrightarrow_{\mathcal{S}} \langle n_1, \sigma_1', s, t_1 \rangle \wedge \\ & \langle 0, \sigma_2, s, \epsilon \rangle \longrightarrow_{\mathcal{S}} \langle n_2, \sigma_2', s, t_2 \rangle \\ & \implies \sigma_1' \approx_L \sigma_2' \wedge t_1 \approx_L t_2 \end{aligned}$$

1

Proof. By cases on the semantic rules for programs

Case S-TICK By induction on the value of T. The base case is $T = 0$, and we

have

$$\langle 0, \sigma_1, s, \epsilon \rangle \longrightarrow_{\mathcal{S}} \langle 1, \sigma_1', s, (1, \sigma_1') \rangle \wedge$$
$$\langle 0, \sigma_2, s, \epsilon \rangle \longrightarrow_{\mathcal{S}} \langle 1, \sigma_2', s, (1, \sigma_2') \rangle$$

where

$$\sigma_1' = \sigma_1[x_1 \mapsto \sigma_1(\bar{x_1})]...[x_n \mapsto \sigma_1(\bar{x_n})]\sigma_2' \quad = \sigma_1[x_1 \mapsto \sigma_2(\bar{x_1})]...[x_n \mapsto \sigma_2(\bar{x_n})]$$

Let $\bar{x_i}$ be some next-cycle symbol such that $x_i \in \{\bar{x_1}, ..., \bar{x_n}\}$. If $\mathcal{T}(\Gamma(\bar{x_i}), \sigma_1) \sqsubseteq L$ then $\mathcal{T}(\Gamma(\bar{x_i}, \sigma_2) \sqsubseteq L$ by Lemma 2. By Lemma 3, $\sigma_1(\bar{x_i}) = \sigma_2(\bar{x_i})$. The same is true for all other next-cycle symbols in $\{\bar{x_1}, ..., \bar{x_n}\}$. Since $\sigma_1' \approx_L \sigma_2$, and σ_2' and σ_1' agree on low symbols $\bar{x_i}$, $\sigma_1' \approx_L \sigma_2'$.

We now consider the general case. We have

$$\langle n, \sigma_1, s, t_1 \rangle \longrightarrow_{\mathcal{S}} \langle n+1, \sigma_1', s, t_1; (n+1, \sigma_1') \rangle \wedge$$
$$\langle n, \sigma_2, s, t_2 \rangle \longrightarrow_{\mathcal{S}} \langle n+1, \sigma_2', s, t_2; (n+1, \sigma_2') \rangle$$

By the induction hypothesis, $\sigma_1' \approx_L \sigma_2'$ and $t_1 \approx_L t_2$. So $t_1; (n+1, \sigma_1') \approx_L t_2; (n+1, \sigma_2')$

Case S-EVAL*:* Follows directly from Theorem 2.

□

2.5.1 Security results with heterogeneously labeled arrays and bit vectors

We now prove that well-typed hardware modules in SecVerilog2 extended with support for heterogeneously labeled arrays and bit vectors that do not contain downgrades also enforce observational determinism. The proof is accomplished by a translation from well-typed programs in the extended language into well-typed programs in the core language which we still refer to as SIRRTL SecVerilog2 vectors are simulated with 1-bit SecVerilog variables and a corresponding translation of SIRRTL environments into SIRRTL environments. SecVerilog2 expressions of width w translate into a vector of w SIRRTL expressions. Assignments of w-bit expressions are unrolled into w assignments. The translation of commands other than assignment statements merely propagate the translation of assignments. The translation is clearly semantics-preserving, and the security result is obtained by showing that the translation is also type-preserving.

The translation splits non-array variables, x, into single-bit representations x_1,...,x_n, where x_i stores the i^{th} bit of the original variable x. The notation $[\![\Gamma; \mathtt{W}; \mathtt{\Theta}]\!]$ denotes the translation of SecVerilog2 environment $\Gamma; \mathtt{W}; \mathtt{\Theta}$ into an SIRRTL environment Γ' containing 1-bit variables. Each variable $x_j \in \Gamma$ is translated into $\mathtt{W}(\mathtt{x_j})$ 1-bit variables whenever $\Gamma(x_j)$ has kind $\mathtt{int} \rightharpoonup \ell$. Otherwise, $\Gamma(x_j)$ has kind $\mathtt{int} \rightharpoonup \mathtt{int} \rightharpoonup \ell$, and x_j represents an array, translated into $n \times \mathtt{W}(\mathtt{x_j})$ 1-bit

variables where n is the declared length of the array:

$$[\![\Gamma; \mathtt{W}; \Theta]\!] = [\![..., x_j : \tau_j, ...; ..., x_j : w_j, ..., ; \tau_j : k_j, ...]\!] \triangleq \{..., X_j, ...\}$$

$$\text{where } X_j = \begin{cases} x_{j,1} : \tau_{j,1}, ..., x_{j,wj} : \tau_{j,wj} & \text{if } k_j = \mathtt{int} \rightharpoonup \ell \\ x_{j,1,1} : \tau_{j,1,wj}, ..., x_{j,n,wj} : \tau_{j,n,wj} & \text{if } k_j = \mathtt{int} \rightharpoonup \mathtt{int} \rightharpoonup \ell \end{cases}$$

The translation of expressions and assignment statements is shown in Figure 2.16. The translation for other commands than statements merely propagates the translation of statements. The translation for both statements and expressions also include Γ, $\mathtt{W}$, and Θ as arguments. For notational convenience, define $\mathtt{W}[\![\mathtt{e}]\!]_{\Gamma;\mathtt{W};\Theta} = \mathtt{w} \iff \Gamma; \mathtt{W}; \Theta \vdash \mathtt{e} : \tau, \mathtt{w}$ for some τ. Each translation of a SecVerilog2 expression produces a vector of SIRRTL expressions. The metasyntax $\mathcal{E}[\![e]\!]_{\Gamma;\mathtt{W};\Theta}(i)$ selects the i^{th} element of the vector produced by $\mathcal{E}[\![e]\!]_{\Gamma;\mathtt{W};\Theta}$. The notation $\vec{e}|_{i\in(n_1,n_2)}$ constructs a new vector by replacing the free occurrences of i in e with each integer in the range (n_1, n_2).

Note that in the rules for logical and arithmetic operators $\mathtt{W}[\![\mathtt{e}_1]\!]_{\Gamma;\mathtt{W};\Theta} = \mathtt{W}[\![\mathtt{e}_2]\!]_{\Gamma;\mathtt{W};\Theta}$. In the rule for the translation of addition and subtraction, C_{n-1}, is a straightforward bit-level representation of the carry-out from the summation of the digits at $n-1$. The translation for arrays produces a nested conditional assignment that dynamically checks the value of the indexing expression (e) to select from among n w-size vectors where n is the declared size of the array and w is the width of each array element. The translation for assignment unrolls the assignment into separate assignments for each of the 1-bit variables. Although not shown, integer constants translate into their binary representations.

Figure 2.17 shows the type-directed translation from SecVerilog2 typing derivations to SIRRTL types. The translation is only defined for types of well-

$$\mathcal{E}[\![x]\!]_{\Gamma;\mathtt{W};\Theta} \triangleq \overrightarrow{x_1, ..., x_{\mathtt{W(x)}}}$$

$$\mathcal{E}[\![e_1 \ \mathsf{bop} \ e_2]\!]_{\Gamma;\mathtt{W};\Theta} \triangleq \overrightarrow{\mathcal{E}[\![e_1]\!]_{\Gamma;\mathtt{W};\Theta}(i) \ \mathsf{bop} \ \mathcal{E}[\![e_2]\!]_{\Gamma;\mathtt{W};\Theta}(i)}\Big|_{i\in(1,\mathtt{W}[\![\mathtt{e_1}]\!]_{\Gamma;\mathtt{W};\Theta})}$$

$$\mathcal{E}[\![e_1 + e_2]\!]_{\Gamma;\mathtt{W};\Theta} \triangleq \overrightarrow{\mathcal{E}[\![e_1]\!]_{\Gamma;\mathtt{W};\Theta}(i) \oplus \mathcal{E}[\![e_2]\!]_{\Gamma;\mathtt{W};\Theta}(i) \oplus C_{i-1}}\Big|_{i\in(1,\mathtt{W}[\![\mathtt{e_1}]\!]_{\Gamma;\mathtt{W};\Theta})}$$

$$\mathcal{E}[\![e_1 - e_2]\!]_{\Gamma;\mathtt{W};\Theta} \triangleq \overrightarrow{\mathcal{E}[\![e_1]\!]_{\Gamma;\mathtt{W};\Theta}(i) \oplus \neg\mathcal{E}[\![e_2]\!]_{\Gamma;\mathtt{W};\Theta}(i) \oplus C_{i-1}}\Big|_{i\in(1,\mathtt{W}[\![\mathtt{e_1}]\!]_{\Gamma;\mathtt{W};\Theta})}$$

$$\mathcal{E}[\![\{e_1, e_2\}]\!]_{\Gamma;\mathtt{W};\Theta} \triangleq \overrightarrow{\mathcal{E}[\![e_1]\!]_{\Gamma;\mathtt{W};\Theta}(i - w_2 + 1)}\Big|_{i\in(w_2,w_1+w_2)} :: \overrightarrow{\mathcal{E}[\![e_2]\!]_{\Gamma;\mathtt{W};\Theta}(i)}\Big|_{i\in(1,w_2)}$$

$$(\text{where } w_1, w_2 = \mathtt{W}[\![\mathtt{e_1}]\!]_{\Gamma;\mathtt{W};\Theta}, \mathtt{W}[\![\mathtt{e_2}]\!]_{\Gamma;\mathtt{W};\Theta})$$

$$\mathcal{E}[\![e \ll n]\!]_{\Gamma;\mathtt{W};\Theta} \triangleq \overrightarrow{\mathcal{E}[\![e]\!]_{\Gamma;\mathtt{W};\Theta}(i - n)}\Big|_{i\in(1,\mathtt{W}[\![\mathtt{e}]\!]_{\Gamma;\mathtt{W};\Theta}-\mathtt{n})} :: \underbrace{0...0}_{\text{n times}}$$

$$\mathcal{E}[\![e \gg n]\!]_{\Gamma;\mathtt{W};\Theta} \triangleq \underbrace{0...0}_{\text{n times}} :: \overrightarrow{\mathcal{E}[\![e]\!]_{\Gamma;\mathtt{W};\Theta}\, i}\Big|_{i\in(n,\mathtt{W}[\![\mathtt{e}]\!]_{\Gamma;\mathtt{W};\Theta})}$$

$$\mathcal{E}[\![x\mathtt{[}e\mathtt{]}]\!]_{\Gamma;\mathtt{W};\Theta} \triangleq \mathcal{E}[\![e]\!]_{\Gamma;\mathtt{W};\Theta} == \mathcal{E}[\![0]\!]_{\Gamma;\mathtt{W};\Theta} \ ? \ \overrightarrow{x_{0,1}, ..., x_{0,\mathtt{W}[\![\mathtt{x[e]}]\!]_{\Gamma;\mathtt{W};\Theta}}} :$$
$$\mathcal{E}[\![e]\!]_{\Gamma;\mathtt{W};\Theta} == \mathcal{E}[\![1]\!]_{\Gamma;\mathtt{W};\Theta} \ ? \ \overrightarrow{x_{1,1}, ..., x_{1,\mathtt{W}[\![\mathtt{x[e]}]\!]_{\Gamma;\mathtt{W};\Theta}}} :$$
$$...$$
$$\mathcal{E}[\![e]\!]_{\Gamma;\mathtt{W};\Theta} == \mathcal{E}[\![n-1]\!]_{\Gamma;\mathtt{W};\Theta} \ ? \ \overrightarrow{x_{n-1,1}, ..., x_{n-1,\mathtt{W}[\![\mathtt{x[e]}]\!]_{\Gamma;\mathtt{W};\Theta}}} :$$
$$\overrightarrow{x_{n,1}, ..., x_{n,\mathtt{W}[\![\mathtt{x[e]}]\!]_{\Gamma;\mathtt{W};\Theta}}}$$

$$(\text{where } n = 2^{\mathtt{W}[\![\mathtt{x[e]}]\!]_{\Gamma;\mathtt{W};\Theta}} - 1)$$

$$\mathcal{E}[\![x = e]\!]_{\Gamma;\mathtt{W};\Theta} \triangleq$$
$$\mathtt{begin}$$
$$\mathcal{E}[\![x]\!]_{\Gamma;\mathtt{W};\Theta}(1) = \mathcal{E}[\![e]\!]_{\Gamma;\mathtt{W};\Theta}(1);$$
$$...;$$
$$\mathcal{E}[\![x]\!]_{\Gamma;\mathtt{W};\Theta}(\mathtt{W}[\![\mathtt{x}]\!]_{\Gamma;\mathtt{W};\Theta}) = \mathcal{E}[\![\mathtt{e}]\!]_{\Gamma;\mathtt{W};\Theta}(\mathtt{W}[\![\mathtt{x}]\!]_{\Gamma;\mathtt{W};\Theta});$$
$$\mathtt{end}$$

Figure 2.16: Translation from SecVerilog2 expressions to SIRRTL expressions.

$$\textsc{TTrans-Const}\ \frac{}{\mathcal{T}[\![\Gamma;\mathtt{W};\Theta\vdash \mathtt{n}:\bot,\mathtt{w}]\!]\hookrightarrow\bot}\qquad \textsc{TTrans-Var}\ \frac{\Gamma(x)=\tau \qquad \mathtt{W(x)}=\mathtt{w}}{\mathcal{T}[\![\Gamma;\mathtt{W};\Theta\vdash \mathtt{x}:\tau,\mathtt{w}]\!]\hookrightarrow \overrightarrow{\mathcal{T}[\![\tau]\!](i)}\Big|_{\mathtt{i}\in(1,\mathtt{w})}}$$

$$\textsc{TTrans-Logical}\ \frac{\begin{array}{c}\Gamma;\mathtt{W};\Theta\vdash \mathtt{e}_1:\tau_1,\mathtt{w} \qquad \Gamma;\mathtt{W};\Theta\vdash \mathtt{e}_2:\tau_2,\mathtt{w}\\ \Theta\vdash\tau_1:\mathtt{int}\rightharpoonup\ell \qquad \Theta\vdash\tau_2:\mathtt{int}\rightharpoonup\ell\\ \tau=i\mapsto(\tau_1\ i)\sqcup(\tau_2\ i)\end{array}}{\begin{array}{c}\mathcal{T}[\![\Gamma;\mathtt{W};\Theta\vdash \mathtt{e}_1\ \mathtt{bop}\ \mathtt{e}_2:\tau,\mathtt{w}]\!]\\ \hookrightarrow\\ \overrightarrow{\mathcal{T}[\![\tau_1]\!](i)\sqcup\mathcal{T}[\![\tau_2]\!](i)}\Big|_{i\in(1,w)}\end{array}}\ (\text{when bop}\in\{\vee\wedge\oplus\})$$

$$\textsc{TTrans-Arith}\ \frac{\begin{array}{c}\Gamma;\mathtt{W};\Theta\vdash \mathtt{e}_1:\tau_1,\mathtt{w} \qquad \Gamma;\mathtt{W};\Theta\vdash \mathtt{e}_2:\tau_2,\mathtt{w}\\ \Theta\vdash\tau_1:\mathtt{int}\rightharpoonup\ell \qquad \Theta\vdash\tau_2:\mathtt{int}\rightharpoonup\ell\\ \tau=i\mapsto\bigsqcup_{j\in(1,i)}((\tau_1\ j)\sqcup(\tau_2\ j))\end{array}}{\begin{array}{c}\mathcal{T}[\![\Gamma;\mathtt{W};\Theta\vdash \mathtt{e}_1\ \mathtt{bop}\ \mathtt{e}_2:\tau,\mathtt{w}]\!]\\ \hookrightarrow\\ \overrightarrow{\bigsqcup_{j\in(1,i)}(\mathcal{T}[\![\tau_1]\!](j)\sqcup\mathcal{T}[\![\tau_2]\!](j))}\Big|_{i\in(1,w)}\end{array}}\ (\text{when bop}\in\{+-\})$$

$$\textsc{TTrans-Concat}\ \frac{\begin{array}{c}\Gamma;\mathtt{W};\Theta\vdash \mathtt{e}_1:\tau_1,\mathtt{w}_1 \qquad \Gamma;\mathtt{W};\Theta\vdash \mathtt{e}_2:\tau_2,\mathtt{w}_2\\ \Theta\vdash\tau_1:\mathtt{int}\rightharpoonup\ell \qquad \Theta\vdash\tau_2:\mathtt{int}\rightharpoonup\ell\\ \tau=i\mapsto \mathtt{if}(i>w2)\ (\tau_1\ i-w_2+1)\ (\tau_2\ i)\end{array}}{\begin{array}{c}\mathcal{T}[\![\Gamma;\mathtt{W};\Theta\vdash\{\mathtt{e}_1;\mathtt{e}_2\}:\tau,(\mathtt{w}_1+\mathtt{w}_2)]\!]\\ \hookrightarrow\\ \overrightarrow{\mathcal{T}[\![\tau_1]\!](i-w_2+1)}\Big|_{i\in(w_1,w_1+w_2)}::\overrightarrow{\mathcal{T}[\![\tau_2]\!](i)}\Big|_{i\in(1,w_2)}\end{array}}$$

$$\textsc{TTrans-LShift}\ \frac{\begin{array}{c}\Gamma;\mathtt{W};\Theta\vdash \mathtt{e}:\tau,\mathtt{w} \qquad \Theta\vdash\tau:\mathtt{int}\rightharpoonup\ell\\ \tau'=i\mapsto \mathtt{if}\ (i>n)\ (\tau\ i-n+1)\ \bot\end{array}}{\begin{array}{c}\mathcal{T}[\![\Gamma;\mathtt{W};\Theta\vdash \mathtt{e}<<\mathtt{n}:\tau',\mathtt{w}]\!]\\ \hookrightarrow\\ \overrightarrow{\mathcal{T}[\![\tau]\!](i-n+1)}\Big|_{i\in(1,w-n)}::\underbrace{\bot\ldots\bot}_{\text{n times}}\end{array}}\qquad \textsc{TTrans-RShift}\ \frac{\begin{array}{c}\Gamma;\mathtt{W};\Theta\vdash \mathtt{e}:\tau,\mathtt{w} \qquad \Theta\vdash\tau:\mathtt{int}\rightharpoonup\ell\\ \tau=i\mapsto \mathtt{if}\ (i>w-n)\ \bot\ (\tau\ i+n)\end{array}}{\begin{array}{c}\mathcal{T}[\![\Gamma;\mathtt{W};\Theta\vdash \mathtt{e}>>\mathtt{n}:\tau,\mathtt{w}]\!]\\ \hookrightarrow\\ \underbrace{\bot\ldots\bot}_{\text{n times}}::\overrightarrow{\mathcal{T}[\![\tau]\!](i+n)}\Big|_{i\in(n,w)}\end{array}}$$

$$\textsc{TTrans-ArrIndex}\ \frac{\begin{array}{c}\Gamma(x)=\tau_x \qquad \mathtt{W(x)}=\mathtt{w}\\ \Gamma;\mathtt{W};\Theta\vdash \mathtt{e}:\tau_\mathtt{e},\mathtt{w}_\mathtt{e}\\ \Theta\vdash\tau_x:\mathtt{int}\rightharpoonup\mathtt{int}\rightharpoonup\ell\\ \Theta\vdash\tau_e:\mathtt{int}\rightharpoonup\ell\\ \tau'_e=i\mapsto\bigsqcup_{j\in(j,w_e)}\tau_e\ j\end{array}}{\begin{array}{c}\mathcal{T}[\![\Gamma;\mathtt{W};\Theta\vdash \mathtt{x[e]}:\tau'_\mathtt{e}\sqcup(\tau_\mathtt{x}\ \mathtt{e}),\mathtt{w}]\!]\\ \hookrightarrow\\ \Big(\bigsqcup_{i\in(1,w_e)}\mathcal{T}[\![\tau_e]\!](i)\Big)\sqcup\mathcal{T}[\![\tau_x\ e]\!]\end{array}}$$

Figure 2.17: Type-directed translation from SecVerilog2 types to SIRRTL types.

typed expressions. The rules have the form

$$\frac{\mathcal{P}_1 \quad \ldots \quad \mathcal{P}_n}{\mathcal{T}[\![\Gamma; \mathtt{W}; \Theta \vdash \mathtt{e} : \tau_{\mathtt{SecVerilog2}}, \mathtt{w}]\!] \hookrightarrow \overrightarrow{\tau_{SIRRTL}}}$$

where $\mathcal{P}_i$ is a premise in a type rule of SecVerilog2, and

$$\mathcal{T}[\![\Gamma; \mathtt{W}; \Theta \vdash \mathtt{e} : \tau_{\mathtt{SecVerilog2}}, \mathtt{w}]\!] \hookrightarrow \overrightarrow{\tau_{SIRRTL}}$$

denotes that the typing derivation surrounded in brackets translates into the vector of SIRRTL types $\overrightarrow{\tau_{SIRRTL}}$, and the meta-syntax $\overrightarrow{\tau_{SIRRTL}}(i)$ denotes the i^{th} element in the vector of types. The notation $\mathcal{T}[\![\tau_{SecVerliogBL}]\!]$ is used for recursive translation of types and it denotes $\overrightarrow{\tau_{SIRRTL}}$ such that

$$\mathcal{T}[\![\Gamma; \mathtt{W}; \Theta \vdash \mathtt{e} : \tau_{\mathtt{SecVerilog2}}, \mathtt{w}]\!] \hookrightarrow \overrightarrow{\tau_{SIRRTL}}$$

where Γ, $\mathtt{W}$, Θ, and e are all always clear from context. The target-language type may contain types mentioned in the inference rules of the derivation.

Note that SIRRTL types are a strict subset of SecVerilog2 types, and that SecVerilog2 types which are not SIRRTL types are abstractions, if-types, and case-types. The if-types and case-types are straightforwardly translated into functions fully-applied to program variables (which are SIRRTL types) so these translations are not shown. Also note that in the translation rules shown in Figure 2.17, all resulting types have kind ℓ and are therefore SIRRTL types.

It is straightforward to check that the SIRRTL program after transformation is semantically equivalent. The soundness result is obtained by showing that the translation is also type-preserving. That is, well-typed SecVerilog2 programs that do not contain downgrading translate into well-typed SIRRTL programs. Since well-typed SIRRTL programs enforce observational determinism,

SecVerilog2 programs share the same result. We now show a proof of the type-preservation result for commands.

In addition to the well-formedness requirements of type environments in SIRRTL, SecVerilog2 environments also require the following for a SecVerilog2 environment to be well-formed:

Definition 2 (Well-Formedness of Environments) *An environment* $\Gamma; \mathtt{W}; \mathtt{\Theta}$ *is well-formed, written* $\vdash \Gamma; \mathtt{W}; \mathtt{\Theta}$*, if for all* $x \in \Gamma$ *such that* $\Gamma(x) = \tau$*,* $\mathtt{W}(\mathtt{x}) = \mathtt{w}$*, and* $\Theta \vdash \tau : \mathtt{int} \rightharpoonup \ell$*, we have* $(\tau\ i)$ *is in the image of* τ *for all* $i \in (1, w)$*. Otherwise,* $\Theta \vdash \tau : \mathtt{int} \rightharpoonup \mathtt{int} \rightharpoonup \ell$ *and there is some* n *such that* $(\tau\ j)$ *is in the image of* τ *for all* $j \in (1, n)$ *and* $(\tau\ j\ i)$ *is in the image of* $(\tau\ j)$ *for all* $i \in (1, w)$*.*

Lemma 1 *If* e *is a SecVerilog2 expression, then for all* $\Gamma, \mathtt{W}, \mathtt{\Theta}$ *such that* $\vdash \Gamma; \mathtt{W}; \mathtt{\Theta}$*,* $\Gamma; \mathtt{W}; \mathtt{\Theta} \vdash \mathtt{e} : \tau, \mathtt{w}$ *,and* $\Theta \vdash \tau : \mathtt{int} \rightharpoonup \ell$*, let* $(\tau\ i)$ *is defined denote that* $(\tau\ i)$ *is in the image of* τ*, then* $(\tau\ i)$ *is defined and of kind* ℓ *for all* $i \in (1, w)$*.*

Proof. By induction over the type rules of SecVerilog2

- T-CONST: trivial.
- T-VAR: by the definition of $\vdash \Gamma; \mathtt{W}; \mathtt{\Theta}$
- T-LOGICAL: By typing rule $\mathtt{W}[\![\mathtt{e}]\!]_{\Gamma;\mathtt{W};\mathtt{\Theta}} = \mathtt{W}[\![\mathtt{e_1}]\!]_{\Gamma;\mathtt{W};\mathtt{\Theta}} = \mathtt{W}[\![\mathtt{e_2}]\!]_{\Gamma;\mathtt{W};\mathtt{\Theta}} = \mathtt{w}\ \Theta \vdash \tau_i \vdash \mathtt{int} \rightharpoonup \ell$ for $i \in \{1, 2\}$. By the induction hypothesis, $(\tau_i\ j)$ is defined for all $j \in (1, w)$ and $i \in \{1, 2\}$. Therefore, $(\tau\ j)$ is defined for all $j \in (1, w)$. By typing rule τ is of kind $\mathtt{int} \rightharpoonup \ell$ and so its application to i is of kind ℓ.
- T-ARITH: similar to T-LOGICAL.

- T-CONCAT: $\mathtt{W}[\![\mathtt{e}]\!]_{\Gamma;\mathtt{W};\Theta} = \mathtt{w}_1 + \mathtt{w}_2$. We show that $(\tau\ i)$ is defined for $i \in (1, w_2)$ and for $i \in (w_2, w_1 + w_2)$. $(\tau\ i) = (\tau_2\ i)$ for $i \in (1, w_2)$ and by the induction hypothesis $(\tau_2\ i)$ is defined and so is $(\tau\ i)$. $(\tau\ i) = (\tau_1\ i - w_2 + 1)$ for $i \in (w_2, w_1 + w_2)$, and since $i - w_2 + 1 \in (1, w_1)$ for $i \in (w_2, w_1 + w_2)$, $(\tau_1\ i - w_2 + 1)$ is defined and so is $(\tau\ i)$. By typing rule τ is of kind $\mathtt{int} \rightharpoonup \ell$ and so its application to i is of kind ℓ.
- T-LSHIFT: $\mathtt{W}[\![\mathtt{e{\ll}n}]\!]_{\Gamma;\mathtt{W};\Theta} = \mathtt{w}$. For the case in which $n < w$ we show that $(\tau'\ i)$ is defined for $i \in (1, n)$ and for $i \in (n, w - n)$. $(\tau'\ i) = (\tau\ i - n + 1)$ for $i \in (n, w + n)$ and since $i - n + 1 \in (1, w)$ $(\tau\ i - n + 1)$ is defined by the induction hypothesis. Otherwise $(\tau'\ i) = \bot$. If $n >= w$, then $(\tau'\ i) = \bot$. By typing rule τ' is of kind $\mathtt{int} \rightharpoonup \ell$ and so its application to i is of kind ℓ.
- T-RSHIFT: Similar to T-LSHIFT.
- T-ARRINDEX: By the definition of $\vdash \Gamma; \mathtt{W}; \Theta$, there is some n such that $(\tau_x\ j\ i)$ is defined for all $j \in (1, n)$ and for all $i \in (1, w)$. Though not shown in the typing rule for conciseness, we require that n is greater than the maximum number expressible in w_e bits, so τ_x is defined over the range from 1 to all possible valuations of e and $\tau_x\ e$ is defined over $(1, w)$ By the induction hypothesis, τ_e is defined over $(1, w_e)$, and so τ'_e is defined and of kind $\mathtt{int} \rightharpoonup \ell$. Since τ_x is applied to e it also of kind $\mathtt{int} \rightharpoonup \ell$, the application of $\tau'_e \sqcup (\tau_x\ e)$ to an i is of kind ℓ.

□

Lemma 2 (Type Preservation of Expressions) *If e is a SecVerilog2 expression, then for all $\Gamma, \mathtt{W}, \Theta$ such that $\vdash \Gamma; \mathtt{W}; \Theta$, $\Gamma; \mathtt{W}; \Theta \vdash \mathtt{e} : \tau, \mathtt{w}$, $\Theta \vdash \tau : \mathtt{int} \rightharpoonup \ell$, and there exists a derivation of $\mathcal{T}[\![\Gamma; \mathtt{W}; \Theta \vdash \mathtt{e} : \tau, \mathtt{w}]\!] \hookrightarrow \overrightarrow{\tau_{SIRRTL}}$, let $\mathcal{E}[\![e]\!]_{\Gamma;\mathtt{W};\Theta} = e_1, ..., e_n$*

and $\overrightarrow{\tau_{SIRRTL}} = \tau_{SIRRTL,1}, ...\tau_{SIRRTL,m}$, *then* $[\![\Gamma; \mathtt{W}; \Theta]\!] \vdash \mathtt{e_1} : \tau_{\mathtt{SIRRTL},1}, ..., [\![\Gamma; \mathtt{W}; \Theta]\!] \vdash \mathtt{e_w} : \tau_{\mathtt{SIRRTL},\mathtt{w}}$

Proof. By induction over the translation rules of expressions using the type translation rules of SecVerilog2, the type rules of SIRRTL, Lemma 1, the definition of well-formedness, and the definition of $[\![\Gamma; \mathtt{W}; \Theta]\!]$. As before, let $(\tau\ i)$ is defined mean that $(\tau\ i)$ is in the image of τ.

- v: by the definition of $[\![\Gamma; \mathtt{W}; \Theta]\!]$, TTRANS-VAR, and Lemma 1.
- e_1 bop e_2: By T-LOGICAL (and T-ARITH) $\mathtt{W}[\![\mathtt{e_1}]\!]_{\Gamma;\mathtt{W};\Theta} = \mathtt{W}[\![\mathtt{e_2}]\!]_{\Gamma;\mathtt{W};\Theta} = \mathtt{w}$, and by Lemma 1, $(\tau_1\ i)$ and $(\tau_2\ i)$ are both defined and of kind ℓ for $i \in (1, w)$. Since e is well-typed, by TTRANS-LOGICAL or TTRANS-ARITH, the typing derivation of e translates into $\overrightarrow{(\mathcal{T}[\![\tau_1]\!](i)) \sqcup (\mathcal{T}[\![\tau_2]\!](i))}\Big|_{i\in(1,w)}$ if bop is logical or $\overrightarrow{\bigsqcup_{j\in(1,i)}(\mathcal{T}[\![\tau_1]\!](j) \sqcup \mathcal{T}[\![\tau_2]\!](j))}\Big|_{i\in(1,w)}$ if bop is arithmetic.In either case Lemma 2 holds of e_1 and e_2 by the induction hypothesis. The typing rule of binary operators in SIRRTL requires that the type of e is the join of the types of e_1 and e_2. For both arithmetic and logical operators the translation is the join of the type of e_1 at index i with the type of e_2 at index i, and so Lemma 2 holds of e.
- $\{e_1, e_2\}$: By Lemma 1, $(\tau_1\ i - w_2 + 1)$ is defined and of kind ℓ for $i \in (w_2, w_1 + w_2)$ and $(\tau_2\ i)$ is defined and of kind ℓ for $i \in (1, w_2)$. The formal language SIRRTL does not support concatenations. However, the typing derivation of e translates into $\overrightarrow{\mathcal{T}[\![\tau_1]\!](i - w_2 + 1)}\Big|_{i\in(w_1,w_1+w_2)} :: \overrightarrow{\mathcal{T}[\![\tau_2]\!](i)}\Big|_{i\in(1,w_2)}$ which is the concatenation of the types of e_1 and e_2 shifted appropriately. By the induction hypothesis, Lemma 2 holds of e_1, and so the bits of e in range $(w_1, w_1 + w_2)$ are well-typed. Similarly, Lemma 2 holds of e_2 and so the remaining bits are also well-typed. Since by the definition of $\mathcal{E}[\![\{e_1, e_2\}]\!]_{\Gamma;\mathtt{W};\Theta}$,

the bits of e_1 and e_2 are concatenated in the same way as their types, each bit of $\mathcal{E}[\![e]\!]_{\Gamma;\mathtt{W};\Theta}$ is well-typed in SIRRTL by the corresponding bit of $\mathcal{T}[\![\tau]\!]$ (by the SIRRTL typing rule of variables and the definition of $[\![\Gamma;\mathtt{W};\Theta]\!]$).

- e'«n, e'»n: The formal language SIRRTL does not support shifts. This case is similar to $\{e_1, e_2\}$, noting that the bits of e which are not bits of e' are always zero, and by the SIRRTL typing rule for constants are well-typed with type $\bot$.
- $x[e]$: By T-ARRINDEX $\mathtt{W}[\![\mathtt{e}]\!]_{\Gamma;\mathtt{W};\Theta} = \mathtt{w_e}$, and the maximum valuation of e is n as defined in the translation rule. By the definitions of $\vdash \Gamma;\mathtt{W};\Theta$ and $[\![\Gamma;\mathtt{W};\Theta]\!]$ and Lemma 1 ($\tau_x\ j\ i$) is defined and of kind ℓ for $j \in (1, n)$ and $i \in (1, w)$, so $P(x_{i,j})$ holds for $j \in (1, n)$ and $i \in (1, w)$. The definition of $\mathcal{E}[\![x[e]]\!]_{\Gamma;\mathtt{W};\Theta}$ evaluates to some SIRRTL variable $x_{i,j}$ so by the definition of $[\![\Gamma;\mathtt{W};\Theta]\!]$ and the typing rule of SIRRTL variables, Lemma 2 holds of $x[e]$.

□

Lemma 3 (Type Preservation) *If c is a SecVerilog2 command, Γ is a type context, and $\mathtt{W}$ is a width environment such that $\Gamma;\mathtt{W};\Theta \vdash \mathtt{c}$, then $[\![\Gamma;\mathtt{W};\Theta]\!] \vdash \mathtt{c}$.*

Proof. By induction on the translation of commands. The only interesting case is assignment. Because the translation for expressions is type-preserving when those expressions have $\mathtt{int} \rightharpoonup \ell$ types (Lemma 2), and all expressions appearing in commands have such types (by assignment rule), and the SecVerilog2 type rule applies the type function for both sides of the assignment to each integer less than the width of the assigned expression, w, then the translation will result in w separate SIRRTL assignments, which will all type-check. □

Theorem 3 (Observational Determinism) *If c is a SecVerilog2 command, Γ is a type context, and $\mathtt{W}$ is a width environment such that $\Gamma; \mathtt{W}; \Theta \vdash: \mathtt{c}$, then c obeys observational determinism.*

Proof. This theorem follows directly from Lemma 3 and the proof of observational determinism for SIRRTL. □

CHAPTER 3

INFORMATION FLOW VERIFICATION OF TRUSTZONE

This chapter discusses the application of SecVerilog to verify the information flow security of an implementation of a simplified, but realistic multi-core prototype of a processor that resembles the ARM TrustZone architecture. As a result of this study, we found it necessary to extend SecVerilog with support for heterogeneously labeled arrays and bit vectors as described in Section 2.2.2 and Section 2.2.1 of Chapter 2. Our experiments suggest that information flow analysis is efficient, and programmer effort is modest. We also show that HDL-evel information flow security can detect hardware vulnerabilities, including several found in commercial processors. Although the processor described in this Chapter does not include timing channel protection, and its implementation includes unconstrained downgrades, this study suggests that HDL-level information flow security can be applied to practical hardware security architectures.

3.1 Background: ARM TrustZone

ARM TrustZone is a representative security architecture that is used widely in practice. Its applications include embedded systems and smartphones. TrustZone uses hardware mechanisms to provide an execution environment that isolates high-security software from low-security software. Other commercial security architectures [17] also aim to provide an isolated execution environment, so we believe the findings of this study are applicable to other architectures as well.

TrustZone partitions the hardware and software into two security domains, called the *secure world* and the *normal (non-secure) world*. The high-security software executes in the secure world, and the remaining software executes in the normal world. The software executing in the normal world is prevented from accessing data owned by the secure world. Practical systems need to allow communication between security domains. For this purpose, TrustZone assumes the secure-world software is trustworthy, and allows it to access data in either world. The threat-model of TrustZone does not address timing channel attacks, and the only physical attacks it addresses are simple ones that exploit debug interfaces.

TrustZone isolates the secure and normal worlds by introducing a security tag, called the *NS bit*. TrustZone uses access control mechanisms in hardware that check the NS bit. Each processing core stores the NS bit in a program status register (PSR) to indicate which world is currently executing on the core. Each bus master and slave that may be used in the secure world is also extended with an NS bit. For example, DMA engines and display controllers may have an NS bit. A core can switch its security domain by executing trusted software called the *monitor mode* which executes with secure world privilige. The monitor mode is entered by an explicit instruction or through interrupts. The normal world cannot change any NS bits.

Access to resources is controlled based on the NS bit in order to isolate the secure world from the normal one. The system (AXI) bus appends the NS bit of the bus master to each transaction. Bus slaves inspect the NS bit and prevent the normal world from accessing secure-world resources. The data of each world is isolated through memory address partitioning, and access controls. For ex-

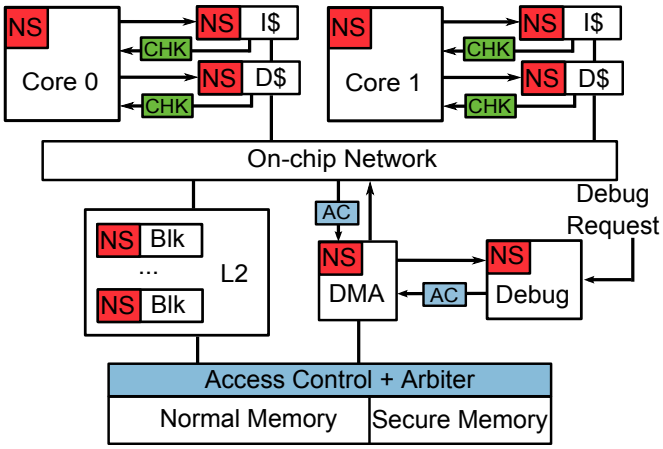

Figure 3.1: TrustZone prototype implementation.

ample, main memory (DRAM) is partitioned into secure and normal based on address ranges, and the NS bit is checked for accesses to the secure-world partition. TrustZone protects debug interfaces by preventing normal-world debug requests from affecting the secure world. Similarly, normal-world accesses to secure-world interrupt configuration registers are disallowed. In some implementations of TrustZone, data from both worlds can coexist in caches. Coexistence is permitted by extending each cache line with an NS bit guarded by access control. Similarly, TLBs can be extended to store address mappings from both worlds.

3.2 Prototype implementation

TrustZone is an architectural specification that can be implemented in many ways. Our prototype is designed to study the practicality of verification with information flow. We implemented key security features for multicore implementations of TrustZone, but did not include non-essential features.

Figure 3.1 shows a block diagram of our implementation of TrustZone. Our implementation includes two five-stage pipelined MIPS processing cores, private L1 caches, a shared L2 cache, a DMA engine, a ring network, and a memory module. Each core has private L1 instruction and data caches, which are connected to a shared L2 cache through a ring network. The L2 cache includes a prefetch buffer. The system includes a DMA engine that can move data between memory locations. The DMA engine takes requests from processing cores through a memory-mapped interface connected to the ring network. It also has an external debug interface. The L2 cache and the DMA engine are connected to the main memory controller through an arbiter. Our processor was implemented in 16,234 lines of Verilog code. To test the functionality of the processor, we used unit tests with over 166 test vectors and executed programs including vector addition, vector multiplication, merge sort, binary search, and a masked filter.

The prototype implements the security features of TrustZone that are necessary to isolate the secure world from the normal world. The processing cores, the DMA engine, and the debug interface include an NS bit to indicate the security domain. The NS bits for the DMA engine and the debug interface can be changed by a secure-world core through a memory-mapped interface. All bus transactions, memory requests, and memory response packets carry the NS bit of the core or DMA engine that initiated the request.

The prototype supports world switches for cores through an instruction. World-switching is implemented in a way that ensures that the NS bits of in-flight instructions are not corrupted. The pipeline is stalled until all in-flight instructions have completed. Then, the NS bit of the core is changed. When

the NS bit changes, SecVerilog clears (sets to 0) registers with security labels that depend on the NS bit. This clearing prevents implicit downgrading [110]. For example, register files and the PC register are cleared. In this design, arguments between the two worlds are passed via memory. A secure-world handler is located at Address 0. The core changes the PC to the location storing a switch-to-normal handler if the PC is 0 in the normal world.

The caches allow data from both worlds to coexist. Each line of the L1 and L2 caches are extended with the NS bit to indicate the world that owns the data. On a hit, the NS bit of the access is compared to the NS bit of the cache line. If there is a mismatch, the access is treated as a miss.

The bus slaves use access control. For example, normal-world requests to the DMA engine are rejected when the DMA engine is in the secure world. The main memory includes an access control module. A partition control register in the module partitions the address space between worlds. The partition control register is memory-mapped and can only be modified by a secure-world request.

The prototype implements the security features of TrustZone necessary to protect the confidentiality and integrity of the secure world in a multi-core SoC. Protection includes support for hardware IP modules (e.g., a DMA engine) and a debug interface. However, the security worlds in TrustZone are orthogonal to traditional privilege levels and virtual memory. Our processor does not include supervisor/user mode or virtual address translation. Also, optional features such as additional peripherals, coherent accelerators, tightly coupled memory, and protected interrupts are not implemented.

TrustZone Security Policy	Information Flow Policy Label	Downgrading
P1. Normal world core/IP cannot **(C)** read or **(I)** write secure-world memory/IP	`CT` for secure world core/IP/memory `PU` for normal world core/IP/memory (dependent type based on the NS bit)	D1-1. Secure world reads/writes to normal-world memory/IP D1-2. Timing dependence (common for all)
P2. **(I)** Normal world cannot change NS bits	`PT` for NS bits	D2-1. Secure world writes to an NS bit D2.2. Legitimate normal-to-secure NS-bit switches
P3. **(I)** Normal world cannot change TrustZone control registers	`PT` for TrustZone control registers	D3-1. Secure-world writes to TrustZone control registers

Table 3.1: Core TrustZone policy expressed as information flow constraints with downgrades. (C) and (I) represent policy for confidentiality and integrity, respectively. IP (Intellectual Property) is a hardware module.

3.3 Representing security requirements as information flow policies

This subsection uses TrustZone as an example to show how HDL-level IFC can verify an isolated execution environment. TrustZone isolates the secure world from the normal world using access control policies and mechanisms that control normal-world accesses. As shown in the first column of Table 3.1, the goal of each access control policy is to protect either the confidentiality (C) or the integrity (I) of security-sensitive state.

The high-level security goal of TrustZone can be expressed as information flow constraints that address either confidentiality or integrity requirements. The confidentiality policies specify that no information can flow from secure-world processing cores, memory, or hardware (IP) blocks to normal-world modules. The integrity constraints ensure that no information from a normal-world core/memory/IP can affect a secure-world core/memory/IP or other trusted

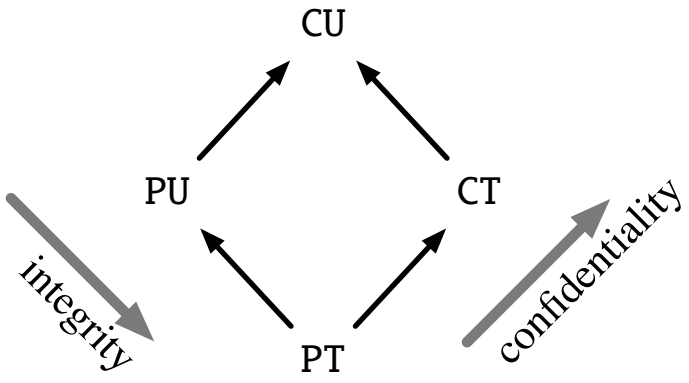

Figure 3.2: Security lattice for TrustZone.

state such as NS bits and TrustZone control registers.

The above information flow constraints can be translated into an information flow policy expressed with a security lattice and labels in HDL code. To express both confidentiality and integrity levels, we define four security levels: `CT`, `CU`, `PT`, and `PU`. The first letter represents the confidentiality level (confidential or public) and the second letter represents the integrity level (trusted or untrusted). Then, we define a security lattice that prevents confidential information from flowing to public and untrusted information from affecting trusted as shown in Figure 3.2. In the figure, the arrows represent the direction of allowed information flow.

The second column in Table 3.1 shows how the TrustZone implementation is labeled using the security levels in the lattice. To protect both confidentiality and integrity of secure-world state, variables in secure-world processing cores, memory, and hardware IP blocks are labeled `CT` while normal-world ones are labeled `PU`. Signals that are statically allocated to one world are annotated with fixed labels. Most hardware resources (e.g., the processing cores) can be switched between the two worlds. The security labels of time-shared modules

use a dependent type and are expressed as a function (`world`) of the NS bit (`ns`) associated with that module. Here, `world(ns)` maps the value of `ns` to a security label: 1 maps to `PU` and 0 maps to `CT`. Signals that must be trusted, but are not confidential, are labeled `PT`. For example, NS bits and TrustZone control registers, such as the one that partitions memory among worlds, must be trustworthy. The clock and reset variables are also labeled `PT`.

Unfortunately, strict noninterference is too restrictive for practical systems like TrustZone because it does not allow any communication between security levels. TrustZone prevents the normal world from acting maliciously, but trusts the secure world to release information to or accept information from the normal world correctly. This permitted communication violates noninterference and causes type errors.

To bridge the gap between noninterference and practical security policies, we introduce declassification and endorsement so that designers can explicitly allow exceptions to noninterference. Declassification releases confidential information to the public. Endorsement changes the security level of untrusted information so that it is considered trusted. The term *downgrading* refers to both.

The third column in Table 3.1 shows how downgrading is used to express the security policy of TrustZone. TrustZone allows the secure world to access the normal world (D1-1). The TrustZone threat model does not include timing-channel attacks, so information flows through timing (D1-2) are allowed. Secure-world writes to NS bits (D2-1) and control registers (D3-1) are allowed even though their values may be read by the normal world. TrustZone allows the normal world to trigger a world switch through a special instruction (D2-2) even though this causes a flow from the normal world to the NS bit. The next

Component Name	$C \rightarrow P$	$U \rightarrow T$
Core Pipeline	$AddrCtrl \rightarrow NSB$ (1) $AddrCtrl \rightarrow AddrCtrl$ (1)	
L2 Cache	$Data \rightarrow CReg$ (1)	
	$AddrCtrl \rightarrow AddrCtrl$ (2) $AddrCtrl \rightarrow NSB$ (12)	
Network	$AddrCtrl \rightarrow AddrCtrl$ (4) $AddrCtrl \rightarrow NSB$ (1)	
DMA Engine	$AddrCtrl \rightarrow NSB$ (1)	
Debug Interface	$AddrCtrl \rightarrow NSB$ (1)	
Memory Arbiter	$AddrCtrl \rightarrow AddrCtrl$ (2)	
Memory Access Control Module	$Data \rightarrow CReg$ (1)	
	$AddrCtrl \rightarrow NSB$ (4)	
Main Memory	$Data \rightarrow Data$ (1)	$Data \rightarrow Data$ (1)
	$AddrCtrl \rightarrow AddrCtrl$ (2)	

Table 3.2: Downgrading expressions in our prototype.

section discusses how downgrading is used in the prototype in more detail.

3.4 Uses of downgrading

Our TrustZone implementation passes type checking when analyzed by SecVerilog. Type-checking formally guarantees that, aside from variables which affect downgraded expressions, there is no violation of the information flow policy expressed in the code. Downgraded expressions explicitly permit exceptions to the policy

Table 3.2 summarizes the uses of downgrading in each microarchitecture component of our implementation of TrustZone. The table categorizes these downgrading expressions as confidentiality exceptions ($C \rightarrow P$), integrity exceptions ($U \rightarrow T$), or both. The downgrade expressions are further classified by the type of variable – data (Data), address or control (AddrCtrl), NS-bit (NS), or control register (CReg) – at the source and destination using the notation `source`

$\rightarrow$`destination`. Address or control variables include valid/ready variables in the network, and instruction decode outputs and stall variables in the cores. The numbers in parenthesis indicate the number of downgrade expressions of that form.

SecverilogBL enforces a timing-sensitive security property; however, the threat model of the architecture under study does not address timing channels. Timing channels will cause type errors even though they are not considered threats. Therefore, timing channels must be distinguished from other kinds of illegal information flows. Categorizing flows based on variable type is useful for doing so; timing channels in hardware typically originate from an address or control variable, but are not directly derived from data.

The following paragraphs summarize why downgrading was needed based on the variable type categorization.

Data $\rightarrow$ Data TrustZone allows the secure world (`CT`) to read or write normal-world (`PU`) memory, contrary to the lattice policy. When secure world writes to normal-world memory, confidentiality is violated. Similarly, secure-world reads from normal-world could violate integrity. The data must be downgraded to permit the intended behavior. This use of downgrading is safe because the secure world is trusted.

Data $\rightarrow$ CReg Control registers are labeled `PT`, because they are used to control both secure-world and normal-world operations. However, control registers can also be modified by the secure-world. This is a violation of the confidentiality policy (since it is a flow from `CT` to `PT`), and must be explicitly permitted with downgrading. Note that normal world is still prevented from setting con-

```
input          {world(ns1)} cpu1_valid;
input          {world(ns2)} cpu2_valid;
input          {PT}         ns1;
input          {PT}         ns2;
output         {PT}         ns_out;
...
if      (cpu1_valid == 1) ns_out <= ns1;
else if (cpu2_valid == 1) ns_out <= ns2;
...
```

Figure 3.3: A flow from control signals to the NS bit due to resource arbitration.

trol registers since downgrading is performed only for a write from the secure world.

AddrCtrl $\rightarrow$ **Data, AddrCtrl** $\rightarrow$ **AddrCtrl** Illegal flows from address/control variables to address, control, and data variables are caused by timing interference between security levels. Timing interference leaks information from address/control variables ($AddrCtrl$) but not data variables ($Data$).

AddrCtrl $\rightarrow$ **NSB** Resources which are used by both worlds cause flows from control variables to the NS bit. Figure 3.3 shows a representative example of this type of flow. It shows a bus arbiter that accepts requests from both cores that could be executing in either world. The output NS bit becomes the NS bit of the core that is granted access. Here, the NS bit is labeled `PT` because its integrity needs to be protected from the normal world, leading to information flow from `CT/PU` to `PT`. In the core, the NS bit is changed by an instruction with label `world(ns)`, similarly requiring downgrading. Here, downgrading affects the timing of the NS bit change, but does not introduce a vulnerability.

The information flow analysis in SecVerilog formally proves timing-sensitive noninterference. However, explicit uses of downgrading expressions are used to weaken noninterference. We argue that in our implementation, downgrad-

ing is used only under the authority of the secure world, and therefore that information release cannot be controlled by the normal world. To show that this is true, we note that information is never downgraded if the secure world never performs an operation. In other words, downgrading can be removed if the secure world is hard-coded to not execute. Both $Data \rightarrow Data$ (secure-world reads/writes to memory) and $Data \rightarrow Creg$ (secure-world writes to control registers) flows happen under an `if` condition that checks if an access is from the secure world. These downgraded information flows never happen if there is no secure-world access. The flows from $AddrCtrl$ variables cause timing contention. Downgrading is unnecessary if there is no secure-world access because `ns_out` will always be 1 (normal world). Since information is never downgraded when the secure world is inactive, this suggests that information release cannot be controlled by the normal world.

3.5 Security bug detection

Here, we study the effectiveness of the proposed information flow analysis at detecting security bugs. We developed a set of security bugs based on reported vulnerabilities in commercial products [97, 98] as well as possible mistakes.

Bugs 1-5: Access Control Omission In TrustZone, access control checks ensure that trusted/confidential state can only be accessed by the secure world. If access control checks are left out, security is violated. We model five bugs, which omit an access control check for 1) the control register that partitions main memory between worlds, 2) the main memory, 3) the debug interface, 4) the L2 cache prefetch buffer, and 5) the L2 cache blocks. Bug 3 is inspired by

```
input          {PT}              ns;
input [dw-1:0] {world(ns)}       data_in;

reg   [dw-1:0] {PT}              part_reg;
    ...
    // Detected bug.
    part_reg <= data_in;
    ...
    // Correct code.
    if (ns == 0) part_reg <= downgrade(data_in, PT);
    ...
```

Figure 3.4: A detected access control omission.

a back door in the Actel ProASIC3 [79], Bug 4 models a vulnerability found in an AMD processor [35], and Bug 5 models a privilege escalation attack in Intel processors that support SMM mode [98]. Figure 3.4 shows how omitted access control checks for the partition register are detected. `data_in` has the label `CT` when `ns` is `0` and `PU` when `ns` is 1. The code on line 7 is detected as a bug because the type system cannot prove that `data_in` is trusted. The correct code on line 10 adds a check to ensure that the `ns` bit is 0, implying that `data_in` is trusted in this context. Bugs 2-4 are detected and fixed similarly.

Bug 5: Cache Poisoning We emulate and detect a subtle vulnerability found in Intel processors [98]. The vulnerability allows a user-mode process to execute arbitrary code in System Management Mode (SMM), the highest privilege level. SMM mode is only used to execute SMM handlers – interrupt handlers requiring such high privilege. In the vulnerable processor, the region of physical memory which stores SMM handlers is protected by access control in the memory interface. A control register can mark this region as un-cacheable, and it does so by default. However, the control register can be modified without SMM privilege, allowing an attacker to make the SMM memory cacheable. Then, the attacker can write to the address of an SMM interrupt handler, and change the

handler code in a cache. Subsequent executions of the handler address will hit in the cache and execute the attacker's code. We modeled this vulnerability in our processor by removing the NS-bit tags and access control from the L2 cache while keeping checks at the memory interface. We added a control register that sets cache-ability for the secure world, which can be modified by either world. The bug is detected because the cache lines can receive data that is from either world, but no access control is present.

Bug 6: NS-bit Flip Memory requests are transmitted with an NS bit that indicates the security level of the request. This bug inverts the NS bit so that a memory request from a normal-world core is interpreted as a secure-world access. This bug is detected because flipping an NS bit changes the type of dependently typed variables. In the network, the input and output data variables both have types that depend on the NS bit. If the input and output NS bits do not match, the security labels of the input and output data will also not match causing an error. Even if the bit is flipped in multiple places, the error will be detected because eventually the input and output types will not match. This demonstrates the benefit of information flow analysis, which tracks the propagation of data throughout the design.

Bug 7: Network Routing Bug This bug models a network implementation that leaks secrets by incorrectly routing a response from the secure-world memory to a normal-world core. In our TrustZone implementation, the bug is prevented by the memory response access control checks and the L1 cache tags. To test the bug, we removed the access checks and used an L1 cache that keeps data from only one security domain at a time. This bug is detected at the interface between the L1 caches and the on-chip network. Without the checks at

the response ports, the type system cannot prove that the NS bit of a response matches the NS bit of a cache.

Bug 8: World Switch Bug For a world switch (i.e., context switch) from normal world to secure world, the processor pipeline must complete all in-flight instructions before changing the NS bit. Otherwise, in-flight normal-world instructions will execute with escalated privilege. We model a vulnerable mode switch by omitting the pipeline drain step. This bug is detected because changing the NS bit causes the labels of the dependently typed registers to change. SecVerilog prevents label changes from leaking information by dynamically clearing register contents.

Bug 9: Memory Address Change Bug To understand the limitations of HDL-level IFC, we constructed two bugs that change the memory address at the memory interface. Figure 3.5 illustrates the bugs. In both cases, the address is changed from the secure-world region to the normal-world region so that a write into secure-world memory gets stored in normal-world memory. This allows the normal world to read data that should be stored in the secure-world memory. Bug 9-1 is not detected, because downgrading allows the secure-world access to write data into the normal-world memory. On the other hand, Bug 9-2 is detected, because the change in the secure-world memory address is triggered by a normal-world variable (`normal_world_trigger`). The examples show that functional bugs in the secure world may lead to undetected bugs through downgrading, but only if there is no influence from the normal world. Vulnerabilities that do not affect downgraded variables are always detected.

Other Bugs Hicks et al. [35] proposed SPECS, a run-time bug detector. They evaluated it by implementing 14 bugs in the OpenRISC processor. The bugs in-

```
// 0x0000-0x8000 is the secure-world memory.
// the rest is the normal-world memory.

// Code common to both bugs
wire [0:31] {world(ns)} addrout, addrin;
wire [0:31] {world(ns)} datain;
reg  [0:31] {CT} data_sec;
reg  [0:31] {PU} data_norm;
always@(*) begin
    if((ns == 0) && (addrout > 'h8000))
        data_norm = downgrade(datain, PU);
    else
        data_sec  = datain;
end
// Bug 9-1: not detected
assign addrout = (addrin <= 'h8000) ?
                 addrin + 'h8000 : addrin;
// Bug 9-2: detected
wire {PU} normal_world_trigger;
assign addrout = (normal_world_trigger ?
                 addrin + 'h8000 : addrin;
```

Figure 3.5: Bug 9: memory address change bugs.

cluded privilege escalation, register target/source redirection, interrupt-register contamination, interrupt disabling, code injection, jump instruction disabling, and others. While we could not implement those bugs in our processor architecture (e.g., our prototype does not have protection rings or an MMU), we reviewed the HDL code studied for SPECS. These bugs all allow a user-mode process to change supervisor-mode variables. Therefore, these bugs should all be detected by information flow analysis if user-mode variables are labeled `PU` and supervisor-mode variables are labeled `CT`.

3.6 Evaluation

Programming Overhead Table 3.3 shows the number of lines of code for the unverified version of our processor (Unverified) and the verified version with se-

Component	Unverified	Verified	Percentage
Top-Level Module	1391	1412	1.5%
Processor	3474	3504	0.86%
L1 Cache	1250	1308	4.6%
Access Control	0	75	N/A
On-chip Network	2122	2557	1.7%
L2 Cache	2976	3093	3.9%
DMA controller	525	549	4.6%
Debug interface	350	369	5.4%
Main memory	974	1015	4.2%
Library Modules	2780	2818	1.4%
Total	16234	16700	2.9%

Table 3.3: Programming overhead (lines of code).

curity labels (Verified). We emphasize that the verification procedure is purely static and performed at compile time. However, the implementation changes slightly 1) to add extra variables specifically for encoding dependent types and 2) to aid the program analysis phase that estimates the run-time values of dependent types. The code increases by 2.9%.

Area, Power, and Performance Overheads The processor was synthesized using Cadence Design Compiler using a standard 90nm library to obtain performance, area, and power results. The clock frequency and CPI were identical for the verified and unverified versions. The area and power overheads are negligible (0.37% and 0.32%).

CHAPTER 4

THE HYPERFLOW PROCESSOR: GENERAL, HARDWARE-ENFORCED INFORMATION FLOW POLICIES

This Chapter presents HyperFlow, a processor architecture and implementation designed for information-flow security that is verified with ChiselFlow at design time, providing strong security assurance about its design and implementation. ChiselFlow is described in Chapter 2. The HyperFlow architecture is carefully designed to remove disallowed information flows between security levels, including timing channels, and the information flows within the design are statically verified using a security type system. HyperFlow is also implemented as an extension of a fully-featured processor that is capable of running an OS.

HyperFlow security policies. Beyond being a practical, realistic processor, HyperFlow also innovates in the security policies it enforces. Unlike prior processors with verified information flow, which only supported simple, fixed 2-point or 4-point policy lattices, HyperFlow can enforce complex application-defined security policies directly in hardware, in line with work on information-flow security in operating systems and programming languages which suggests that real applications need rich lattice policies that can capture complex trust relationships among mutually distrusting principals [66, 24, 104, 14].

We show how to encode complex and dynamic security policies involving both confidentiality and integrity even for applications built from communicating processes serving mutually distrusting principals. By enforcing security policies in hardware, the software component of the trusted computing base is minimized, and strong security assurance is obtained. While practical tagged

architectures have previously been built with the ability to encode information flow policies [106, 22], they do not handle timing channels, and their implementations have not been secured with an information flow HDL.

HyperFlow represents rich lattice policies in hardware by introducing *hypercube labels*, in which software-level labels are mapped to points in a hypercube. Hypercube labels enable efficient comparisons between security levels directly in hardware, and they are amenable to static checking in the security-typed HDL.

Controlled downgrading. To be practical, systems based on information-flow security must allow exceptions to noninterference [33]. For example, applications must be able to release the results of computing on secrets. This can be accomplished by *downgrading*, a mechanism for relaxing information-flow policies. Uncontrolled downgrading is dangerous, so the HyperFlow ISA provides instructions for controlled downgrading. Downgrading of confidentiality policies (declassification) is permitted only when it is *robust* [101]—secrets can be released only if the downgrade can be influenced only by the owners of these secrets. Dually, downgrading of integrity policies (endorsement) is permitted only it when it is *transparent* [10]: that is, if the party providing the endorsed data could have read it. Together, these conditions ensure that information flow is *nonmalleable*. Nonmalleable information flow is enforced not only at the ISA level but also at the HDL level, providing similarly strong assurance about the implementation.

Secure interprocess communication. Another novel and challenging feature of HyperFlow is its support for secure communication across trust domains. HyperFlow allows but constrains interproness communication (IPC) via

shared memory. It also supports the secure communication via registers for passing arguments and return values on system calls. System calls and shared function libraries present another challenge that HyperFlow addresses—both scenarios require a mechanism by which untrusted code can invoke trusted code. HyperFlow introduces an information-flow secure *call gate* [76, 96] mechanism to make cross-domain control transfers secure.

Memory protection. Conventional systems use virtual memory to isolate pages that belong to different applications. However, hardware support for virtual memory is complex and its correctness also depends on other mechanisms such as cache coherence, which is notoriously difficult to implement correctly. HyperFlow replaces conventional memory protection with *security tags* associated with each physical page (or frame) of memory. Security tags are mapped to hypercube labels using a mapping defined by the operating system; accesses to memory are then mediated using hypercube labels. The security of this mechanism is checked in the HDL code at design time. Despite its novel mechanism for memory protection, HyperFlow also provides a virtual-memory interface to support existing operating systems and applications.

Prototype implementation. We implement HyperFlow as an extension to the RISC-V Rocket processor, and HyperFlow supports a complete RISC-V ISA. Our implementation allows conventional virtual-memory protection to interoperate with HyperFlow's information flow protection. HyperFlow implements performance-critical features that were absent in prior processors secured with an IFC HDL. The prototype implementation shows that the new security features in HyperFlow add moderate area overhead, largely due to the additional storage for security tags, and moderate performance overhead due to timing-

channel protection. The HyperFlow implementation is also more fully-featured than prior information-flow secured processors. The timing-safe implementations of these features also type-checks with ChiselFlow, providing strong assurance that the implementation is secure.

The rest of this chapter is organized as follows. Section 4.1 describes the security policies enforced by HyperFlow. Section 4.2 describes the instruction set architecture of HyperFlow. Section 4.3 describes the microarchitecture of our prototype implementation of HyperFlow and how the implementation is labeled and type-checked with ChiselFlow. Section 4.4 evaluates the performance, power, and area of our prototype implementation as well as the expressiveness of our ISA. Section 4.5 describes implementation and design trade-offs as well as the process of implementing information-flow secure hardware.

4.1 Security Policies in HyperFlow

HyperFlow enforces information flow security policies directly in hardware. Prior work on label models for information flow security support rich policies allowing mutually distrusting principals to communicate [66, 24, 104, 14]. These label models represent policies using lattices of information flow labels. HyperFlow can enforce policies described in these models because it can enforce general lattice-based policies.

4.1.1 Confidentiality and integrity policies

Information flow labels in HyperFlow support reasoning about both confidentiality and integrity. An information flow label $\ell = (c, i)$ in HyperFlow is a pair of a confidentiality level c and an integrity level i. Confidentiality and integrity levels in HyperFlow both form lattices that are ordered by $\sqsubseteq_C$ and $\sqsubseteq_I$ respectively. The ordering on confidentiality levels specifies constraints on secrecy; if $c \sqsubseteq_C c'$, then c is no more confidential than c'. Similarly, if $i \sqsubseteq_I i'$, then i is at least as trustworthy as i'. The ordering of integrity levels and confidentiality levels is dual: high confidentiality levels are more restrictive than low ones, whereas low integrity levels are more restrictive than high ones. The orderings on confidentiality and integrity levels are lifted to a lattice of labels $\sqsubseteq$; if $c \sqsubseteq_C c'$ and $i \sqsubseteq_I i'$ then $(c, i) \sqsubseteq (c', i')$. We write $C(\ell)$ and $I(\ell)$ to denote just the confidentiality or integrity part of the label respectively.

4.1.2 Lattices via bit vectors

In order to support efficient computations and comparisons of labels in hardware, HyperFlow represents lattices over bit vectors. We first explain the ordering of confidentiality levels. Levels are mapped to a point in a hypercube, which is expressed using a bit vector. A bit vector b is split into $d \in D$ dimensions, each of K bits. Bit vectors are then functions from $[1, D]$ to $[0, 2^K - 1]$, and the notation $b(i)$ represents the value in the i^{th} dimension of b. Bit vectors b_1 and b_2 are ordered in the confidentiality lattice, written $b_1 \sqsubseteq_C b_2$ if each dimension of b_1 is numerically less than or equal to the corresponding element of b_2. The join ($\sqcup_C$) and meet ($\sqcap_C$) of two confidentiality components are respectively computed by

$$
\begin{aligned}
b \in B &= [1, D] \to [0, 2^K - 1] \\
b_1 \sqsubseteq_C b_2 &\triangleq \forall d \in [1, D]. b_1(d) \leq b_2(d) \\
(b_1 \sqcup_C b_2)d &\triangleq \max\{b_1(d), b_2(d)\} \\
(b_1 \sqcap_C b_2)d &\triangleq \min\{b_1(d), b_2(d)\}
\end{aligned}
$$

Figure 4.1: Confidentiality ordering over bit vectors.

$$
\begin{aligned}
b \in B &= [1, D] \to [0, 2^K - 1] \\
b_1 \sqsubseteq_I b_2 &\triangleq \forall d \in [1, D]. b_1(d) \geq b_2(d) \\
(b_1 \sqcup_I b_2)d &\triangleq \min\{b_1(d), b_2(d)\} \\
(b_1 \sqcap_I b_2)d &\triangleq \max\{b_1(d), b_2(d)\}
\end{aligned}
$$

Figure 4.2: Integrity ordering over bit vectors.

taking the maximum or the minimum over the corresponding dimensions of each vector. The lattice over bit vectors is defined more formally in Figure 4.1. As an example, if b_1 and b_2 are each bit vectors of 4, 2-bit dimensions, and b_1 is `10100111` and b_2 is `10010010`, then $b_2 \sqsubseteq_C b_1$. It is straightforward to check that this order has lattice properties (i.e., it is a transitive, reflexive, and antisymmetric partial order). The ordering in the integrity lattice is exactly dual to the ordering in the confidentiality lattice as shown in Figure 4.2.

We write $(c, i) \sqcup (c', i') \triangleq (c \sqcup_C c', i \sqcup_I i')$ and $(c, i) \sqcap (c', i') \triangleq (c \sqcap_C c', i \sqcap_I i')$ to denote the join and meet over labels respectively. We use $\top$ and $\bot$ to denote a sequence of all `1`'s and all `0`'s respectively. In the confidentiality order, $\top$ and $\bot$ are completely secret and completely public respectively. In the integrity order, $\top$ and $\bot$ are completely trusted and completely untrusted respectively. The labels $(\bot, \top)$ and $(\top, \bot)$ are the least and most restrictive labels in information flow order ($\sqsubseteq$).

Other representations of lattices in computer systems have been studied in the past [32]. Because HyperFlow uses information flow labels for access control and timing-channel protection, lattice comparisons and computations need to be done throughout the implementation, and the ability to efficiently update and compare labels directly in hardware is particularly important in designing a processor with strong information flow security. Prior representations of lattices such as adjacency lists and matrices are less space-efficient. Other approaches that rely on caching require software intervention on each lattice operation. The hypercube lattice is most similar to the skeletal representation, also known as the Fidge and Mattern vector clock [40]. However, vector clocks have not been used to represent lattices in hardware in prior work.

4.1.3 Non-malleable downgrading

Systems for information flow control are often intended to enforce noninterference, which prevents all information flows that violate lattice policies. However, noninterference is too restrictive for practical systems. For example, data computed using secrets may eventually need to be released publicly. Noninterference may be weakened through *downgrading* which relaxes information flow labels. Downgrading that weakens confidentiality is said to *declassify* whereas downgrading that weakens integrity is said to *endorse* [103].

Because downgrading weakens noninterference, effort has been made to constrain downgrading to limit its potential to cause harm [75]. In this work, we permit communication that weakens noninterference as long as the downgrading it causes is nonmalleable [10]. Nonmalleable information flow subsumes

two security conditions, robust declassification and transparent endorsement. These security conditions have not been enforced by previous hardware mechanisms.

Robust declassification [101] only permits information to be downgraded by parties that have authority over that information. As in prior work on defining robust declassification [15, 10], authority (trust, privilege) is represented by integrity; only a principal at least as trusted as $I(p)$ can declassify data with confidentiality $C(p)$. This constraint is useful for decentralized systems such as microkernels. A principal A can declassify its data to a principal B, and as long as B does not have integrity $I(A)$, B can observe A's data but is prevented from releasing it elsewhere.

In HyperFlow, a process with label ℓ_{cur} can declassify a label ℓ to ℓ' only if the following holds:

$$C(\ell) \sqsubseteq_C C(\ell') \sqcup_C \big(I(\ell_{cur}) \sqcup_I I(\ell)\big).$$

This condition follows directly from prior work on defining robust declassification in the context of programming languages [15, 10]. Roughly, it allows the confidentiality $C(\ell)$ of the data being declassified to be "made up for" by the integrity $I(\ell)$ of the data being declassified and the integrity $I(\ell_{cur})$ of the current process. In other words, low-integrity processes cannot influence whether or not secrets are declassified. When ℓ can be robustly declassified to ℓ' by a process with label ℓ_{cur}, we write $\ell \xrightarrow[\ell_{cur}]{C} \ell'$. Notably, the condition includes a confidentiality join involving an integrity label component; this is well-defined because $\sqcup_C$ is an operation defined over bit vectors and it is agnostic to whether the vectors represent integrity or confidentiality values.

The dual of robust declassification is transparent endorsement [10]. It sets a

maximum confidentiality on endorsements to prevent opaque writes that could enable attacks. A write is opaque if a principal could have written data but not read it. In HyperFlow, a process with label ℓ_{cur} can endorse a label ℓ to ℓ' if,

$$I(\ell) \sqsubseteq_I I(\ell') \sqcup_I \big(C(\ell_{cur}) \sqcup_C C(\ell)\big).$$

This condition follows directly from work on defining transparent endorsement for a functional programming language [10]. When ℓ can be transparently endorsed to ℓ' by a process with label ℓ_{cur}, we write $\ell \xrightarrow[\ell_{cur}]{I} \ell'$. When $\ell \xrightarrow[\ell_{cur}]{I} \ell'$ and $\ell \xrightarrow[\ell_{cur}]{C} \ell'$ we say that ℓ can be nonmalleably downgraded to ℓ' by a process with label ℓ_{cur} and we write $\ell \xrightarrow[\ell_{cur}]{} \ell'$ [10].

4.2 The HyperFlow Architecture

HyperFlow is realized as a tagged architecture where security labels are explicitly represented as hardware tags for a process, registers, and memory pages. HyperFlow compliments conventional memory protection enforced by virtual memory with security tags that are associated with each physical page (or frame) of memory. Tagged physical memory enables static checking of information flow with a type system. Virtual memory does not ensure noninterference; it is possible for the same physical page to be mapped to virtual addresses owned by distrusting processes. Even if the mapping did ensure noninterference, it would not be possible to prove that noninterference is established purely by inspecting the hardware design, since the mapping is software-defined. The tagged physical memory can also be used to reduce the software trusted computing base by removing the need to rely on virtual memory for process isolation.

The security tags in HyperFlow are information flow labels. By enforcing information flow labels in hardware, HyperFlow can permit isolation among multiple principals that are mutually distrusting, yet communicate. Noninterference precludes communication among mutually distrusting principals, so the information that they are communicating must be downgraded. However, HyperFlow constrains these downgrades by ensuring that they are nonmalleable [10]. In doing so, HyperFlow ensures that processes cannot leak information that they do not have authority over. Enforcing non-malleability requires the ability to inspect the integrity and confidentiality of the information being downgraded as well as the principal initiating the downgrading. This is accomplished by making the information flow labels visible in hardware.

4.2.1 Process levels

Processes executing in HyperFlow are associated with a level, ℓ_{cur}. The level $C(\ell_{cur})$ represents the highest level of secrecy that the process can observe, and $I(\ell_{cur})$ represents the most trusted level of information it can affect. In order for the currently executing process to read a page of memory, m, we require $\mathcal{M}_\ell(m) \sqsubseteq \ell_{cur}$, where $\mathcal{M}_\ell$ is a mapping from pages to their information flow labels. Similarly, to write to m, we require that $\ell_{cur} \sqsubseteq \mathcal{M}_\ell(m)$.

HyperFlow also associates security labels with registers to facilitate two kinds of communication that are needed in processors: 1) communication between userspace applications and the operating system during system calls, and 2) interprocess communication in memory. During system calls, arguments and return values are communicated between the application and system call han-

dler via registers. HyperFlow permits communication using registers by associating labels with registers and through instructions that downgrade registers' labels. Assuming the application is untrusted, the trusted call handler can endorse the registers storing the arguments after inspecting them. At the end of the system call, the call handler can declassify the registers storing the return values before re-entering the public application.

Because information flow labels are used to enforce security, HyperFlow must ensure that the labels accurately reflect the security of the data they protect. General-purpose register r has security label ℓ_r. Normally, to store the content of r to an address in m, we require $\ell_r \sqsubseteq \mathcal{M}_\ell(m)$. Similarly, loading a word from m into r requires $\mathcal{M}_\ell(m) \sqsubseteq \ell_r$.

Though secure, these invariants sometimes prevent necessary communication among distrusting principals. HyperFlow permits interprocess communication among distrusting principals via shared memory so that it provides a familiar software interface. Writes to and reads from shared memory that would violate noninterference require downgrading. Pages can be downgraded; however, downgrading an entire page is too imprecise for many applications. HyperFlow supports downgrades at the granularity of an individual word with downgrading load and store instructions. These instructions work just like conventional loads and stores except that they downgrade the source data while it is copied. HyperFlow also supports page downgrades for zero-copy sharing of entire pages.

Processes in HyperFlow are also protected against information flow violations caused by the instructions that are fetched by the currently executing process. The active process should not execute low-integrity instructions because

this would allow adversaries that the process does not trust to influence the code that the process executes. Similarly, branching conditions that depend on secrets can cause secrets to be released through the instructions that HyperFlow executes. Information leaks through control flow are called *implicit flows*.

HyperFlow prevents implicit flows because ℓ_{cur} also represents the security label of the fetched instruction and control flow decisions Branches cannot depend on a register r unless $\ell_r \sqsubseteq \ell_{cur}$. Similarly, for all instructions that write to a register r, HyperFlow requires $\ell_{cur} \sqsubseteq \ell_r$ to ensure that the label of ℓ_r accurately reflects the process that influenced it.

4.2.2 Information-flow call gates

The restriction on branch conditions and on writes to registers together prevent an untrusted or secret process from invoking code that is trusted or public. However, untrusted applications need to be able to call trusted code when making system calls, and secret applications need to be able to call public functions from libraries. HyperFlow securely supports control transfers of this form with a mechanism that is analogous to call gates that originated in Multics [76]. Call gates in HyperFlow tightly couple the entry point (program counter) that initiates the code with an information flow label that represents the privilege level of that code. A process at level ℓ_{cur} can register a call gate at level ℓ' as long as $\ell_{cur} \sqsubseteq \ell'$. Another process can then invoke a call gate, at which point the program counter is set to the entry point of the gate and ℓ_{cur} is set to the level at which the gate was registered. To allow protected returns from call gates, invoking a call gate also pushes the previous program counter value and level

of ℓ_{cur} onto a hardware stack. The executing process can then invoke a return instruction to pop the stack, jumping to the old pc value and privilege level.

Call gates in HyperFlow are unique in that conventional hierarchical privilege levels are replaced with more general lattice-model information flow labels. By generalizing privilege levels, HyperFlow securely supports control transfers with fewer privilege changes than in a conventional processor while simultaneously providing more fine-grained separation of privilege. For example, in a system managed by a microkernel running on HyperFlow, a network driver can register a call gate at a security level ℓ_{net} that is incomparable with other components of the microkernel. When an application wishes to send a packet over the network, it can directly invoke the call gate transferring immediately to ℓ_{net}. In a conventional processor, the network driver can either run in supervisor mode, in which case the application must implicitly trust the entire kernel, or the network driver can run in userspace. In the second case, the application must first make a system call causing a transition to supervisor mode before the kernel delegates to the userspace driver. In this case, the application must both trust the kernel to delegate to the driver, and there is a performance penalty because of the extra privilege changes.

Using just a single level, ℓ_{cur}, for a given process is often sufficient for applications – particularly legacy applications that do not use information flow labels internally. However, other applications require the ability to operate on data within a space of information flow labels. To permit flexibility with the label of executed instructions, HyperFlow allows the active process to move the level of ℓ_{cur} within a space of labels bounded by ℓ_{lwr} and ℓ_{upr}. When setting the value of ℓ_{cur}, we require $\ell_{lwr} \sqsubseteq \ell_{cur} \sqsubseteq \ell_{upr}$. For a process executing with a

space of labels, $C(\ell_{upr})$ and $I(\ell_{upr})$ represent the most secret and most trusted information that the process can observe and affect respectively. On the other hand, $C(\ell_{lwr})$ and $I(\ell_{lwr})$ represent the least secrecy the process can claim it has observed and the least trustworthy information that it can be influenced by.

4.2.3 Instruction set extensions

HyperFlow introduces new instructions as well as new control and status registers. Security levels in HyperFlow are represented as a pair of confidentiality and integrity components as described in Section 4.1. Levels ℓ_{lwr}, ℓ_{cur}, and, ℓ_{upr} are each stored in control and status registers (CSRs) and are accessed with conventional CSR instructions. The registers that store ℓ_{lwr} and ℓ_{upr} define the bounds for a process. To prevent a process from circumventing its own bounds, the bounds can only be modified when the processor is in the most public and trusted level, that is $\ell_{cur} = (\bot, \top)$. However, ℓ_{cur} can be modified as long as $\ell_{lwr} \sqsubseteq \ell_{cur} \sqsubseteq \ell_{upr}$.

The new instructions are summarized in Table 4.1. The first column describes the instruction name and operands, the second column describes restrictions that must be satisfied when executing instructions, and the third column describes what the instruction does if the restrictions are satisfied. We overload the notation r to denote the value stored in register r. As before, ℓ_r denotes the label associated with r and $\mathcal{M}_\ell(a)$ denotes the label of the page containing address a.

The instructions `DECLREG` and `ENDOREG` downgrade registers. The `DECLREG` instruction declassifies the value stored in r_1 to the confidentiality level stored

Instruction	Restrictions	Behavior
`DECLREG R1, R2`	$\ell_{cur} \sqsubseteq (r_2, I(\ell_{r_1}))$ $\ell_{r_1} \xrightarrow[\ell_{cur}]{C} (r_2, I(\ell_{r_1}))$ $\ell_{r_2} \xrightarrow[\ell_{cur}]{} (\bot, \top)$	$C(\ell_{r_1}) \leftarrow r_2$
`ENDOREG R1, R2`	$\ell_{cur} \sqsubseteq (C(\ell_{r_1}), r_2)$ $\ell_{r_1} \xrightarrow[\ell_{cur}]{I} (C(\ell_{r_1}), r_2)$ $\ell_{r_2} \xrightarrow[\ell_{cur}]{} (\bot, \top)$	$I(\ell_{r_1}) \leftarrow r_2$
`RSTLREG R1`	None.	$\ell_{r_1} \leftarrow \ell_{cur}$ $r_1 \leftarrow 0$
`LWDWN R2, IMM(R1)`	$\mathcal{M}_\ell(r_1 + \texttt{IMM}) \xrightarrow[\ell_{cur}]{} \ell_{r_2}$ $\mathcal{M}_\ell(r_1 + \texttt{IMM}) \xrightarrow[\ell_{cur}]{} \ell_{cur}$	$r_2 \leftarrow \mathcal{M}(r_1 + \texttt{IMM})$
`SWDWN R2, IMM(R1)`	$\ell_{r_2} \sqcup \ell_{cur} \xrightarrow[\ell_{cur}]{} \mathcal{M}_\ell(r_1 + \texttt{IMM})$	$\mathcal{M}(r_1 + \texttt{IMM}) \leftarrow r_2$
`SETMEM R2, IMM(R1)`	$\ell_{cur} = (\bot, \top)$	$\mathcal{M}_\ell(r_1) \leftarrow r_2$ $\mathcal{M}(r_1) \leftarrow 0$
`DECLMEM R2, IMM(R1)`	$\ell_{cur} \sqsubseteq (r_2, I(\mathcal{M}_\ell(r_1 + \texttt{IMM})))$ $\mathcal{M}_\ell(r_1 + \texttt{IMM}) \xrightarrow[\ell_{cur}]{C} (r_2, I(\mathcal{M}_\ell(r_1 + \texttt{IMM})))$ $\ell_{r_2} \xrightarrow[\ell_{cur}]{} (\bot, \top)$ $\ell_{r_1} \xrightarrow[\ell_{cur}]{} (\bot, \top)$	$C(\mathcal{M}_\ell(r_1)) \leftarrow r_2$
`ENDOMEM R2, IMM(R1)`	$\ell_{cur} \sqsubseteq (C(\mathcal{M}_\ell(r_1)), r_2)$ $\mathcal{M}_\ell(r_1 + \texttt{IMM}) \xrightarrow[\ell_{cur}]{I} (C(\mathcal{M}_\ell(r_1 + \texttt{IMM})), r_2)$ $\ell_{r_2} \xrightarrow[\ell_{cur}]{} (\bot, \top)$ $\ell_{r_1} \xrightarrow[\ell_{cur}]{} (\bot, \top)$	$I(\mathcal{M}_\ell(r_1)) \leftarrow r_2$
`REGLGATE R1, R2`	$(\ell_{cur} \sqcup \ell_{r_1} \sqcup \ell_{r_2}) \sqsubseteq r_2$ $\ell_{r_2} \xrightarrow[\ell_{cur}]{} (\bot, \top)$ $\ell_{r_1} \xrightarrow[\ell_{cur}]{} (\bot, \top)$	$T[r_1] \leftarrow r_2$
`LCALL IMM`	None.	$S \leftarrow S :: (\texttt{pc} + \texttt{4}, \ell_{cur}, \ell_{lwr}, \ell_{upr})$ $\texttt{pc} \leftarrow \texttt{pc} + \texttt{IMM}$ $\ell_{cur} \leftarrow T[\texttt{pc} + \texttt{IMM}]$
`LCALL IMM(R1)`	$\ell_{r_1} \xrightarrow[\ell_{cur}]{} (\bot, \top)$	$S \leftarrow S :: (\texttt{pc} + \texttt{4}, \ell_{cur}, \ell_{lwr}, \ell_{upr})$ $\texttt{pc} \leftarrow r_1 + \texttt{IMM}$ $\ell_{cur} \leftarrow T[r_1 + \texttt{IMM}]$
`LRET`	None.	$(pc, \ell_{cur}, \ell_{lwr}, \ell_{upr}) \leftarrow \texttt{tail(S)}$ $S \leftarrow \texttt{head(S)}$
`SETBOUNDS`	$\ell_{cur} = (\bot, \top)$	$\ell_{cur} \leftarrow \ell_{ncur}$ $\ell_{lwr} \leftarrow \ell_{nlwr}$ $\ell_{upr} \leftarrow \ell_{nupr}$

Table 4.1: New Instructions Added in HyperFlow.

in r_2, but it permits the declassification only if it is robust. The first restriction prevents implicit flows by ensuring that ℓ_{cur} can write to the new level of r_1. The second restriction ensures that r_1 can be robustly declassified from ℓ_{r_1} to $(r_2, I(\ell_{r_1}))$.

The last restriction is more subtle – it prevents potential information flow violations that might be caused by the use of r_2 as an argument. The register labels and memory labels are fully public and fully trusted. In most work on secure information flow, labels are public and trusted; otherwise, merely inspecting the labels releases information, and if the labels are not trusted, it is hopeless to rely on them for security. Because this instruction allows the value stored in r_2 to influence a label, it must be permitted to influence fully public and trusted data. A natural way to ensure this is to simply require that $\ell_{r_2} = (\bot, \top)$. However, this restriction would often require extra instructions to first downgrade r_2 before downgrading r_1. Instead, we enforce a less restrictive, but equally secure condition—it must be possible to downgrade r_2 to $(\bot, \top)$ using robust declassifications and transparent endorsements. This relaxed restriction does not weaken security because when the restriction on the label of r_2 holds, it is always possible to first downgrade the label of r_2.

The instruction `RSTREG` allows a process to reclaim a register without downgrading by setting the level of the register r_1 to ℓ_{cur}. In order to avoid possibly downgrading the value stored in r_1, r_1 is cleared. Because this instruction takes no arguments other than r_1, and it happens unconditionally, no restrictions on this instruction are necessary. This instruction is useful for easily resetting the labels of registers because it does not impose any restrictions.

The `LWDWN` instruction works like a normal load word instruction, but it relaxes the restrictions on information flow labels. It permits the load if the value of the page from which the data is loaded could be downgraded to the label of the destination register, and to ℓ_{cur}. Similarly, `SWDWN` works like a store instruction that permits the store if the register contents could be downgraded to

the label of the destination page. Both instructions are useful for interprocess communication via shared memory.

The memory levels can be reset by totally trusted and public software through a `SETMEM` instruction, which takes two arguments: the page-aligned physical address in register r_1 and confidentiality and integrity components in r_2. `SETMEM` sets $\mathcal{M}_\ell(m)$ to the value stored in r_2. The `SETMEM` instruction can only be executed when $\ell_{cur} = (\bot, \top)$. Trustworthy software that uses this instruction should clear the contents of the page to prevent implicit downgrades.

Entire pages can also be declassified/endorsed by user-space applications through the `DECLMEM` and `ENDOMEM` instructions, which are similar to `SETMEM` except that they require the changes in memory levels to be robust/transparent as in `DECLREG` and `ENDOREG`. As with `DECLREG` and `ENDOREG`, information flow violations through labels are also prevented by requiring that the arguments that influence labels can be downgraded securely.

The `REGLGATE` instruction registers a new call gate with a `pc` value of r_1 and a label of r_2 by adding it to a table T, that stores call gates by mapping pc values to labels. The first restriction, $(\ell_{cur} \sqcup \ell_{r_1} \sqcup \ell_{r_2}) \sqsubseteq r_2$, checks that the process creating the gate and the arguments from which the gate is constructed are no more secret and are at least as trusted as the label of the gate. The entries in the call gate table are public and trusted (though the labels of individual gates may be more restrictive), because processes that attempt to use call gates must be able to see whether or not they exist. Therefore, the last two restrictions check that the active process can downgrade the register arguments to public and trusted because they influence the creation of a call gate entry.

The LCALL and LCALLR instructions execute a call gate and have the same instruction formats as conventional JAL and JALR instructions. The LCALL instruction specifies the call-gate entry point with an immediate that is added to the current pc value, whereas the LCALLR instruction specifies the entry point by adding an immediate to a register argument. For both instructions, if the specified entry point is found in the call gate table, the address of the instruction following the call and the value of ℓ_{cur} prior to the call are pushed to a hardware stack S for return addresses and labels. The processor then sets the pc value to the entry point of the gate and sets ℓ_{cur} to the label of the gate. The instruction LRET pops the stack S and returns to the most recent pc value and label.

Finally, the SETBOUNDS instruction permits software that is fully public and fully trusted to set the label bounds. There are CSRs called ℓ_{ncur}, ℓ_{nlwr}, ℓ_{nupr} that privileged software can read and write normally. The SETBOUNDS instruction atomically copies these CSRs to ℓ_{cur}, ℓ_{lwr}, and ℓ_{upr} respectively in a single cycle. This instruction is necessary, because writing to an individual bound register might otherwise temporarily violate the invariant, $\ell_{lwr} \sqsubseteq \ell_{cur} \sqsubseteq \ell_{upr}$.

4.2.4 Semantic changes to existing instructions

In addition to the new instructions, HyperFlow also changes the semantics of existing instructions in order to ensure that the policies described by the information flow labels are enforced. To enforce these polices, a set of invariants must hold for each instruction that is executed. The invariants depend on the kind of instruction being executed. For example, different invariants hold for arithmetic instructions and memory instructions. Table 4.2 summarizes these invariants.

Instruction Type	Invariant
Load instructions	$\ell_{r_a} \sqcup \mathcal{M}_\ell(m) \sqcup \ell_{cur} \sqsubseteq \ell_{r_d}$ $\wedge \mathcal{M}_\ell(m) \sqsubseteq \ell_{cur}$
Store instructions	$\ell_{r_a} \sqcup \ell_{r_v} \sqcup \ell_{cur} \sqsubseteq \mathcal{M}_\ell(m)$
Execute unit	$\ell_{r_{s1}} \sqcup \ell_{r_{s2}} \sqcup \ell_{cur} \sqsubseteq \ell_{r_d}$
Value-dependent branches	$\ell_{r_{s1}} \sqcup \ell_{r_{s2}} \sqsubseteq \ell_{cur}$
All instructions	$\mathcal{M}_\ell(m_i) \sqsubseteq \ell_{cur}$

Table 4.2: Instruction invariants enforced by HyperFlow.

The invariants serve two purposes: 1) to implement memory protection, and 2) to ensure that the labels of the registers and memory pages accurately capture the secrecy and integrity of the data they protect.

Memory protection is enforced by ensuring that when a process with label ℓ_{cur} loads from a page m, $\mathcal{M}_\ell(m) \sqsubseteq \ell_{cur}$, to prevent reads that would violate security. This is explicitly enforced on loads. On stores to m, we require $\ell_{cur} \sqsubseteq \mathcal{M}_\ell(m)$ which is subsumed by the invariant enforced by store instructions listed in the table.

For all instructions regardless of type, HyperFlow must enforce the condition $\mathcal{M}_\ell(m_i) \sqsubseteq \ell_{cur}$, where m_i is the memory page where the instruction is fetched from. This prevents information from leaking to the process via the fetched instruction.

The rest of the invariants preserve the accuracy of the information flow labels. For load instructions, condition

$$\ell_{r_a} \sqcup \mathcal{M}_\ell(m) \sqcup \ell_{cur} \sqsubseteq \ell_{r_d}$$

must also hold, where r_a is the source register that contains the base address

and m is the page which contains the data being loaded. This invariant ensures that the level of the destination register accurately reflects the level of the data it stores. Similarly, store instructions require

$$\ell_{r_a} \sqcup \ell_{r_v} \sqcup \ell_{cur} \sqsubseteq \mathcal{M}_\ell(m)$$

where r_v is the register that contains the value being written, and m is the page being written to. This invariant ensures that the policy described by the level of the page being written to is also not violated by the data being written to the page or by the address.

Computation instructions such as arithmetic and logical instructions, multiplication and division, and floating-point instructions, perform a computation on arguments and write the result back into a destination register. For these instructions,

$$\ell_{r_{s1}} \sqcup \ell_{r_{s2}} \sqcup \ell_{cur} \sqsubseteq \ell_{r_d}$$

must hold, where r_{s1} and r_{s2} are the source registers and r_d is the destination register. The data is influenced by the values of both the source registers (which are bounded by $\ell_{r_{s1}}$ and $\ell_{r_{s2}}$) as well as the process causing the instruction to execute (which is bounded by ℓ_{cur}).

Value-dependent control-flow instructions such as conditional branches must enforce the invariant

$$\ell_{r_{s1}} \sqcup \ell_{r_{s2}} \sqsubseteq \ell_{cur}.$$

Because ℓ_{cur} represents the security level of the current control flow (program counter) as well as the security level of a process, a change to the program counter can only be affected by information that can flow into ℓ_{cur}. This invariant prevents branch instructions that would violate the information flow policy.

When the restrictions on the new HyperFlow instructions or the invariants on existing instructions are violated, there are multiple ways of preventing insecure flows. A natural choice is to cause exceptions that can be handled by software. However, causing exceptions can, in general, also cause information flow violations [63]. To avoid the posibility of nested exceptions, the prototype implementation of HyperFlow converts instructions that would otherwise cause information flow violations into `NOP` instructions.

4.3 HyperFlow Microarchitecture and Labeling

This section describes the microarchitecture extensions necessary for HyperFlow and how they are labeled in the secure HDL. The HyperFlow instruction set architecture can be realized by many implementations and microarchitectures. Here, we discuss our prototype implementation based on the RISC-V Rocket Chip processor. The HyperFlow implementation includes many microarchitecture features such as a pending store buffer, pipelined caches, branch prediction, virtual memory, and atomic memory operations, which were not studied in previous information-flow-secured processor designs.

Every signal in the HDL implementation has a label, though many of them are inferred. The ISA-visible security tags of registers and memory locations and ℓ_{cur} that have already been described in Section 4.2 also represent type system level information flow labels in ChiselFlow. The remaining type-system level labels that protect other signals in the implementation must be consistent with the ISA-visible labels.

HDLs for information flow control prevent information flow violations

through explicit changes in data values such as the storage of a value in a register or memory location, and through the timing of events such as the assertion of a valid or ready signal. In the remaining discussion in this section, we coarsely separate labels in the HDL syntax into *data labels* that protect values, and *timing labels* that protect the timing of events. Examples of data labels include register labels and bypass value labels. The valid bits of data cache entries have timing labels. We also note that this is a coarse characterization; timing flows cannot be cleanly separated from other kinds of information flows in hardware. For example, the values of the data operands of a conventional multiplier influences the time that the multiplication is done executing. We also note that not all labels in the type system are represented by separate physical signals in the hardware as many are functions of the same signal.

Section 4.5 provides more details on the design trade-offs we that considered to make the HyperFlow implementation pass the information flow security analysis performed by the type system, and how the process of implementing hardware with a security-typed HDL differs from that of conventional hardware designs.

4.3.1 Labels in the core and label bypassing

In the processing core, the security label of the current process (ℓ_{cur}) is stored in a new control status resister (CSR). The confidentiality and integrity components, $C(\ell_{r_i})$ and $I(\ell_{r_i})$, of general purpose registers r_i are stored in register banks adjacent to the registers. These label registers can be modified only by the `DECLREG` and `ENDOREG` instructions, which are guarded by logic that checks the

non-malleability conditions, and the RSTLREG instruction which can only set the label of a register to ℓ_{cur}.

The HyperFlow core supports data bypassing. To function correctly, the security labels must be bypassed with the data. For immediate values, the bypassed label is ℓ_{cur}. For a value from a register or a cache, its label travels with the bypassed data. The bypassed labels are themselves labeled with ℓ_{cur} because they might be stalled or updated by the current process.

4.3.2 Memory protection and labels

The memory page labels ($\mathcal{M}_\ell(m)$) are stored in an on-chip table that maps page numbers to labels. The current security label (ℓ_{cur}) is attached to each memory transaction so that information flow and downgrading can be checked in the memory system. When returning data from the memory, the label of the page being read is fetched from the table and appended to the memory response transaction. The label of the accessed data is used to enforce the invariants pertaining to load instructions. Write transactions that modify data in memory are similarly appended with a label that protects the data being stored. The label of this data payload is generated from the processing core initiating the request. The label of the data in a write transaction is compared against the label of the destination page which is stored in the memory label table. Write transactions that would cause information flow violations are dropped.

For a system with large off-chip memory, it may be preferable to store the memory labels in memory along with the data. We have also implemented and tested an alternative design that stores memory labels off chip. In this alterna-

tive design, memory labels are stored in a reserved region of memory that is not exposed to software as part of the available physical address space. When servicing a memory transaction, the state machine stores the transaction in a buffer, and issues an additional request to fetch the memory labels. Once the labels have been fetched, the original memory transaction is issued unless it was a write transaction that would cause an information flow violation.

There is a trade-off between these two implementations; storing labels in memory is more efficient in terms of area because DRAMs are more space-efficient than SRAMs. However, the state machine issues a second memory transaction to fetch labels for every last-level cache miss, effectively doubling the last-level cache miss penalty. Naturally, labels can be stored in an additional label cache between the last-level cache and the memory to reduce the cache miss penalty and the cost of additional area. However, this cache may introduce new timing channels. Because memory transactions are issued by software that executes with label ℓ_{cur}, the labels of the data payloads are ℓ_{cur} and are not in general public. Therefore, label cache accesses might cause timing channels in the same way as conventional caches. The label cache would therefore need to be flushed on a label change or partitioned. Our prototype implementation of in-DRAM labels does not include a label cache, and our off-chip memory is relatively small, so we use the memory label table for our experiments.

Initially, every page is mapped to the most public and trusted label $C(\bot) \land I(\top)$. The `SETMEM`, `DECLMEM`, and `ENDOMEM` instructions issue memory transactions that modify the labels. In the table-based implementation these instructions update the table, whereas the state-machine implementation issues additional memory transactions. The table and state-machine implementations are

boot-strapped differently. In the table-based implementation, the table entries are simply initialized to the public and trusted label on reset. When labels are stored in-DRAM, the labels cannot be reset. Instead, an on-chip bootrom must explicitly issue instructions to initialize each label of memory.

Mapping pages to public and trusted initially is secure in our threat-model because the software that runs initially is trusted and public, and we do not intend to address physical attacks. A design in which memory pages are initially public and untrusted is also possible, and offers some defense against off-chip memory that has been tampered with physically. If the initial memory labels are all public and untrusted, the initial instructions cannot be stored in memory because ℓ_{cur} is initially public and trusted, and executing untrusted instructions would clearly cause an information flow violation. Instead, the initial software needs to be stored in an on-chip bootrom that is public and trusted.

Our prototype implementation of HyperFlow provides memory protection at the granularity of memory pages. However, alternative implementations might provide protection at smaller granularity, such as individual words or cache lines (64B) of memory.As with any hardware-enforced memory protection mechanism, there is a trade-off between protection granularity and memory overhead [96]. By providing distinct labels for each word, HyperFlow could support pointer-based data structures that are labeled heterogeneously. Fine-granularity protection could also be used to thwart attacks that are common in C programs such as buffer overflows by preventing untrusted data from influencing trusted instructions [82, 20]. For our prototype implementation, we chose to use page-granularity labels to reduce memory overhead. Changing the granularity of protection does not change the overall approach or impose challenges

to type-checking. Prior architectures for dynamic information flow tracking propose a multi-granularity protection approach [82, 106] in which fine-grained tags are used when needed, but a single tag can be applied to the entire page when they are not needed. A multi-granular approach would apply naturally to the HyperFlow architecture, though it is less clear if type-checking would be straightforward because the policy can change at run-time.

4.3.3 Cache labels

In the data cache, a data label is added to each cache line to track the memory label for the physical address stored in the cache line. The memory label is appended to a cache refill transaction from memory. The data cache in our prototype is blocking, so memory tags are always brought into the cache before any data is modified or returned to the core. For a load, the cache only returns data if the data label of the accessed cache line flows to ℓ_{cur}. The core updates the destination register only if the label of the returned cache data flows to the label of the destination register. The security of a store is enforced by checking that the label of a pending store buffer entry flows to the label of the cache line, and that the label of a memory transaction flows to the memory page label.

The `LWDWN` instruction may be used to load a value with downgrading and performs three non-malleability checks. The cache checks that the value can be downgraded to ℓ_{cur}, which an input signal to the cache. Then, the core checks that the data label of the cache response can be downgraded to the destination register's label; two checks are added, one near the bypassed data and the other near the register file write-back.

4.3.4 Timing-channel protection

ℓ_{cur} is also used as a timing label to prevent timing channels through microarchitectural state. That is, $C(\ell_{cur})$ is an upper bound on the level of secrecy that the process is permitted to observe by measuring timing. Any microarchitectural state that influences the timing of instructions is protected by ℓ_{cur}. Cache entries, in-flight instructions and cache transactions, translation-lookaside buffer (TLB) and page table walker (PTW) entries, and branch predictor state are examples of state that influences instruction timing. Because the security type system in ChiselFlow enforces timing-sensitive non-interference, timing channels must be removed for the hardware to type-check.

When the value of ℓ_{cur} moves downwards in the lattice, the level of secrecy that the process can observe is decreasing. HyperFlow must prevent secrets owned by the previous level of ℓ_{cur} from leaking to the new one. The processor pipeline is drained to prevent high instructions from stalling low instructions as well as other subtle timing channels through register bypassing. In-flight transactions in pipelined caches are also drained when ℓ_{cur} is lowered.

The pending store buffer in the data cache also introduced a subtle timing channel that we did not initially expect. Outstanding cache-write requests in the pending store buffer are serviced opportunistically when there is no in-flight read request. The store buffer can cause a stall either when the content of the buffer might have a read-after-write hazard or when the buffer is full. To prevent a timing channel, we enforce that all entries of the pending store buffer must have the same label, and the buffer is drained before lowering ℓ_{cur}.

Caches may also cause timing channels when they are shared among secu-

rity levels. For instruction caches, the timing channel can be removed by simply clearing and invalidating cache lines when lowering ℓ_{cur}. However, in the data cache, dirty cache lines must be written back when they are evicted, and cannot be simply invalidated. In our implementation, we require software to issue a cache flush instruction to write-back dirty cache blocks before executing an instruction to lower ℓ_{cur}. When ℓ_{cur} is lowered, the data cache is invalidated in a single clock cycle. Note that the flush is required for correctness, but not security.

While our prototype implementation uses flushing to remove cache timing channels, cache partitioning can also be used to lower the flushing overhead on a label change. With partitioned caches, each partition can have a register for its own security label. Then, the logic for a cache read only searches partitions with labels ℓ_P such that $\ell_P \sqsubseteq \ell_{cur}$. When the security label of a partition changes downwards in the lattice, only that partition will need to be invalidated.

HyperFlow has both *branch prediction*, which predicts whether or not branches are taken, as well as *branch target prediction*. The branch target predictor (BTB) in HyperFlow is fully-associative. The branch history table (BHT) has two-bit states per index and a global history register. Prior work has demonstrated that both forms of branch prediction create timing channels that are capable of leaking secrets from Intel SGX enclaves [46]. To prevent timing channels, when ℓ_{cur} moves downward, the BTB is invalidated and cleared, the BHT is cleared, and the global history register is reset.

4.3.5 Virtual memory

The HyperFlow implementation includes support for virtual memory. While HyperFlow protects memory using memory labels, the virtual memory system provides a familiar interface with a view of private and contiguous memory and permits legacy application software to run on HyperFlow unmodified. Virtual memory support includes instruction and data TLBs as well as a hardware page table walker (PTW). TLBs influence timing because they are caches of recently used Level-1 (L1) page table entries (PTEs). L1 PTEs store mappings from virtual to physical addresses. The PTW serves as a cache of L2 PTEs, which store pointers to L1 PTEs. The TLB and PTW state are labeled with ℓ_{cur}, and the state is cleared when ℓ_{cur} moves downward in the lattice.

Because the TLB and PTW state is labeled with ℓ_{cur}, PTEs must be stored in a memory page with a label that flows to ℓ_{cur}. This restriction must be satisfied by the software that manages the page tables. One simple option is to label the memory pages for page tables with $(C(\bot), I(\top))$, which is the least-restrictive information-flow label. The page table can also be stored in pages with more restrictive labels as long as they flow into the processes that use the page table.

4.3.6 Atomic memory operations

Like the baseline processor it extends, HyperFlow supports atomic memory operations (AMOs). AMOs are critical for operating system implementations because they are needed to implement synchronization primitives. AMOs are implemented by buffering both operands into different slots of the pending store buffer in the data cache before buffering the result back into the first slot of

the pending store buffer. Implementing AMOs securely is challenging because the operands of the AMO arrive at the buffer over multiple clock cycles, meaning the operands might have different timing labels, and because the operands might also have different data labels. Because both AMO operands are buffered-in by the same instruction, both operands have the same timing label. To reduce the number of ports in the AMO execution unit, both operands, and therefore the output, are required to have the same label.

4.4 Evaluation

We developed a prototype implementation of HyperFlow as an extension to a single-core configuration of the RISC-V Rocket Chip processor. This is a more fully-featured processor than those previously implemented with statically checked information-flow [51, 50, 110, 29]. The prototype implementation label-checks with ChiselFlow and successfully runs all of Rocket Chip's ISA and application unit tests.

4.4.1 Processor features

The processor is pipelined, with branch prediction and branch target prediction. The branch history table has 2 bits of state per entry and a global history register. The branch target predictor is fully associative. The execution units include an ALU, a multi-cycle multiplier, and a floating-point unit (FPU).

The FPU is implemented as an independent coprocessor that receives instructions from the main processor, but it has its own independent instruction-

decode unit, floating-point register file, and pipeline. The FPU sends requests to the memory hierarchy independently of the main processor.

The processor has four hierarchical protection rings, and a 32-bit virtual address space divided into 4KB pages. The baseline processor has L1 instruction and data caches each with 64 sets and 4 ways and 64B cache blocks. Both L1 caches have 2 pipeline stages. The data cache has a two-slot pending store buffer. Both caches are virtually indexed and physically tagged. The caches include cache controllers. Separate instruction and data TLBs store level-1 page table entries for each cache. A single hardware page-table walker refills both TLBs on misses and caches recently-used level-2 page-table entries.

Many of these micro-architectural features are absent in prior information-flow secured processor implementations. To the best of our knowledge, HyperFlow is the first to include TLBs, a PTW, branch and branch target prediction, and a pending store buffer. Most of these features introduce subtle timing channels that we address, and needed to address in order to satisfy the type system. HyperFlow is also the first to include data bypassing with fine-grained information flow labels. This necessitates dynamic label bypassing, which we must also label-check. The prototype implementation of HyperFlow includes all of the aforementioned features as well as the ISA and microarchitectural extensions described in Sections 4.2 and 4.3. The HyperFlow prototype does not include hardware accelerators and relies on a hard-wired memory controller on an FPGA for offchip DRAM accesses.

4.4.2 Developer effort

HyperFlow was implemented by a single developer over a period of eight months. While the existing Chisel implementation of the RISC-V Rocket chip provides most of the hardware functionality described above, it contains many timing channels. The majority of the development effort was spent on modifying the microarchitectural design to eliminate these timing channels, especially in the instruction and data cache pipelines, virtual memory hardware, and multiplier. Aside from timing channels, some other hardware features, such as bypassing, required rewriting of the HDL code to make it more amenable to program analysis. The labeling support provided by ChiselFlow was essential for removing timing channels correctly. By contrast, it was straightforward to extend the architecture with new instructions for managing labels, downgrading, and call gates – label-checking these features did not pose significant challenges beyond those of conventional hardware implementation. Label-checking many other features already present in RISC-V imposed no additional effort for label-checking; for example, the instruction expansion units and additional decode logic for compressed instructions label-check trivially.

Label inference significantly reduced required developer effort. However, inferred labels were initially difficult to debug because they are often less intuitive than those that a human would write explicitly. Feedback on inference led to changes to the ChiselFlow label implementation to improve the debug-ability of inferred labels.

	When is Information Downgraded	Number of Downgrades	What is Downgraded
1	On Reset	1	Register tags (for initialization)
2	FPU to Int instructions	1	Values copied from the FPU to integer registers
3	ℓ_{cur} lowers	2	Presence of one outstanding finish transaction
4	Memory instructions	1	Address is downgraded to ℓ_{cur}
5	CSR file writes	1	Data written to CSR file is downgraded to ℓ_{cur}
6	DECLREG, ENDOREG	7 (1 + 3 each)	Register contents, control signals, arguments
7	LWDWN	2	RF writeback data, dcache bypass data
8	SWDWN	1	P-store buffer data
9	DECLMEM, ENDOMEM	9 (1 + 4 each)	Page contents, control signals, arguments
10	RSTLREG	1	Control signal
11	REGLGATE	8	Control signals, arguments, pipelined data labels
12	LCALL, LCALLR	3	Control signal, arguments, pc value
13	LRET	1	Control signal
14	MMIO Responses	1	MMIO response transaction data

Table 4.3: Uses of Downgrades in HyperFlow.

4.4.3 Uses of downgrades

The RTL code for HyperFlow performs downgrades at various points; these downgrades are checked for nonmalleability by the ChiselFlow type system. Our formal results imply that insecure information flows can only arise because of these downgrades, which should therefore be inspected carefully. All but one downgrade is statically checked to be nonmalleable by the type system. Table 4.3 summarizes the uses of RTL-level downgrades. The first column shows the ISA-visible event to which the downgrade is tied, the second column states the number of downgraded expressions in the RTL code, and the third gives a brief description of what is downgraded. For all downgrades other than downgrades of data caused by explicit downgrade instructions, both an endorsement and a declassification happen.

We expand upon these descriptions here. When the processor resets (1), the register file tags are all initialized to $(\bot, \top)$, and the contents are initialized to zero. This initialization requires explicit writes to the tags because the register file labels are implemented as a sequential memory that can be synthesized as

a BRAM block on an FPGA. However, this initialization is secure because the processor is initially public and trusted and boots public and trusted code. This downgrade is an artifact of the prototype implementation and not fundamental to the architecture. The BRAMs used in the implementation require explicit initialization, necessitating this downgrade. By contrast, flip-flops could simply reset to the intended values.

For convenience, copies from the FPU to the integer register file (2) are automatically downgraded if the labels of the data coming out of the FPU can be nonmalleably downgraded. When ℓ_{cur} moves downwards in the lattice (3), it is possible for a single outstanding cache coherence transaction to remain in a pending transaction buffer, causing timing interference. We resolve this with a downgrade, but nothing is leaked if the software is written as described in Section 4.2: prior to lowering ℓ_{cur}, the software should issue a cache flush instruction to flush any buffered coherence transaction. When a memory transaction is issued (4), the data used to compute the address is downgraded to ℓ_{cur} because the address affects the timing of the cache transaction; this downgrade is for convenience because the address can otherwise be downgraded with an instruction. The label of the address is still protected by the data label, and so the store invariant in Table 4.2 is enforced by the data label. To permit the use of performance counters, writes to the CSR file (5) are downgraded to ℓ_{cur}.

The downgrading instructions (`DECLREG`, `ENDOREG`, `LWDWN`, `SWDWN`, `DECLMEM`, and `ENDOMEM`) downgrade the stored data and the arguments to the instructions (6–9). These downgrades are done under a conditional statement that checks that these values are downgraded nonmalleably. As described in Section 4.2, the arguments are also downgraded to $(\bot, \top)$ because the arguments influence

changes to public and trusted labels – this downgrade is also guarded by a non-malleability check. The labels of the arguments are also bypassed, and bypassed labels are labeled ℓ_{cur}. Because the bypassed labels are inspected by the non-malleability check, which influences whether or not the downgrade happens, the labels of the bypassed labels are also downgraded from ℓ_{cur} to $(\bot, \top)$. The control signals that induce the downgrades are also downgraded to $(\bot, \top)$ – this downgrade is always non-malleable because these control signals are labeled ℓ_{cur}. The `LWDWN` instruction downgrades the data in two places in the core: the bypass data from the cache and the register file write-back data from the cache. The `SWDWN` instruction downgrades the stored data from the label in the pending store buffer to the label indicated by the memory tags that are stored in the cache. Neither `LWDWN` nor `SWDWN` changes the valuation of any labels, so these instructions do not induce downgrades of control signals or arguments.

Similarly, for instructions `RSTLREG`, `REGLGATE`, `LCALL`, `LCALLR`, and `LRET` (10–13), control signals are downgraded because these instructions affect public and trusted state. The `REGLGATE` instruction also includes a non-malleability check on pipelined labels. The `LCALL` and `LCALLR` instructions store the old `pc` value in a public and trusted stack, so the `pc` is downgraded from ℓ_{cur} to $(\bot, \top)$.

Finally, one downgrade endorses and declassifies data from memory-mapped IO devices (14). This downgrade is not in general non-malleable because we do not provide protection for or from memory-mapped IO devices.

4.4.4 Uses of dynamic checks

Dynamic checks are alternatives to downgrades. They are dynamic label comparisons that prevent an information flow violation, are expected to never be violated at runtime, and convert an information flow violation into a correctness violation. They are preferable to downgrades because they do not weaken security. However, care must be taken because dynamic checks should only be used when it is expected that the invariant can never be violated. Dynamic checks are used in HyperFlow to establish that $\ell_{lwr} \sqsubseteq \ell_{cur}$. This invariant is established in the control and status register (CSR) file where those registers are stored. However, the fact that this invariant is true is not visible in other components outside the CSR file. Another use of dynamic checks is to prevent timing channels caused by floating-point computation. Because the FPU computes on register values, and the time taken to finish a floating-point computation is data-dependent, the stall signals from the FPU are also data-dependent. The pipeline register stall signals in HyperFlow are labeled with ℓ_{cur}. We use a dynamic check that hides the stall signals from the FPU when the data values do not flow to ℓ_{cur}. Another dynamic check is used to convince the type system that the bypass value from the data cache does not cause timing channels; this dynamic check forces the bypass value from the cache to 0 if the timing label from the data cache response does not flow to ℓ_{cur}, but permits the actual data value to be returned otherwise. In practice, this dynamic check does not cause a functional error because when ℓ_{cur} is lowered, the data cache pipeline is stalled and cannot emit responses. Both the regular data cache bypass value and a downgraded bypass value produced by `LWDWN` are covered b the dynamic check.

4.4.5 RTL synthesis results

We synthesized the baseline processor and HyperFlow using Vivado v2016.2 targeting the 7z020clg484-1 FPGA found on the Zedboard Zynq 7000 development board. The baseline processor uses 34,508 LUTs (64.9%) on the FPGA, whereas HyperFlow uses 40,205 LUTs (75.6%), a LUT utilization overhead of 16.5%. The baseline processor uses 13 (9%) of the block RAM tiles whereas HyperFlow utilizes 19.5 (14%). The majority of the utilization overhead is due to the security tags stored with each cache entry, the tag table that associates tags with memory pages, and dynamic label comparisons, which are used for either access controls or dynamic checks. For both the baseline processor and HyperFlow, Vivado is able to meet a target clock frequency of 25MHz. For both designs, the critical path is through the FPU multiplier, so we expect that the minimum clock period is the same for both designs.

4.4.6 CPI Results

Although HyperFlow has no clock frequency overhead, there is performance overhead for timing channel protection. We measured the cycles per instruction (CPI) for HyperFlow when executing the RISC-V benchmark suite compared to the baseline Rocket Chip processor. For the HyperFlow processor, the processor executes with the same security level during the entire execution of the program. The results are summarized in Table 4.4. HyperFlow incurs a performance penalty because unlike Rocket Chip, the multiplier unit always executes in the worst case number of cycles. This performance penalty can be removed by disallowing multiplications of operands with data labels that do not flow to

Benchmark name	HyperFlow CPI	RISC-V CPI	Percent Overhead
mm	1.089	1.063	2.4%
spmv	1.748	1.678	4.2%
median	1.631	1.284	27%
multiply	1.899	1.115	69%
qsort	1.542	1.531	0.7%
towers	1.052	1.030	2.14%
vvad	1.161	1.094	6.12%
dhrystone	1.206	1.187	1.6%

Table 4.4: Performance results

ℓ_{cur}. The mm benchmark is matrix-matrix multiply, spmv is double-precision sparse matrix vector multiply, median is a median filter, multiply does multiplications, qsort does quicksort on an array of integers, towers solves a towers of Hanoi puzzle, vvad adds two vectors, and dhrystone is the classic synthetic benchmark. The benchmark with the highest overhead is multiply, naturally. The geometric mean overhead is 12.4%.

HyperFlow also introduces performance overhead through hardware state that is flushed or invalidated on label changes, but these occur infrequently—when the active process changes via a context switch. The time between context switches is on the order of tens of milliseconds, so this overhead should be amortized over execution.

4.4.7 Usability

HyperFlow intends to support necessary communication among mutually distrusting principals in an environment managed by an operating system. HyperFlow also supports the expressive information flow label models that have been proposed for prior operating systems and languages for information flow con-

trol. In this section, we demonstrate that HyperFlow supports shared memory interprocess communication, communication through registers for system calls, and enforcement of rich information flow policies.

We demonstrate how labels represented in the FLAM [2] model can be expressed as hypercube labels and enforced. The flow-limited authorization model (FLAM) is a recent model that supports decentralized security policies [2]. To illustrate the usability of the HyperFlow architecture, we implemented a simple application with a decentralized information flow control (DIFC) policy expressed in FLAM. DIFC policies allow communication among mutually-distrusting principals [66, 24, 104, 14].

The application emulates a tax-preparation service where a user ("Bob") sends data to a tax preparer and gets the result back. Both the tax preparer and the user are distrusting. Even though the tax-preparer process is allowed to perform computation on the user's data, HyperFlow prevents it from sharing the user's data or any values derived from it to any party other than the user. In our implementation, the tax-preparer process and the user process communicate through shared memory via IPC. Both processes are managed by trusted software implemented as a single system call that manages labels for the two parties. The application is implemented as assembly that runs in RTL simulations of our information-flow verified processor prototype. This result suggests that the HyperFlow ISA and prototype are sufficient to enforce complex application-defined information flow control policies with IPC and system calls.

Section 4.4.7 provides background on FLAM. Section 4.4.7 demonstrates how FLAM labels can be represented with the hypercube labels of HyperFlow. Section 4.4.7 demonstrates how IPC works in HyperFlow, and Section 4.4.7

demonstrates how system call handlers can be implemented. Section 4.4.7 discusses how we use the IPC and system call primitives to construct the tax preparation application.

Background: The FLAM Label Model and Downgrading

FLAM unifies authorization and information flow polices. Principals p can delegate to each other; given principals p and q, if p *acts for* q, written $p \succeq q$, then p trusts q. Compound principals can be constructed from primitive principals. The conjunctive principal $p \wedge q$, denotes the combined authority of both p and q. Similarly, the disjunctive principal, $p \vee q$, represents the authority of either p or q. Principals together with $\succeq$ form a lattice, and $p \wedge q \succeq p \succeq p \vee q$ for any p, q.

In FLAM, principals are also information flow labels. The confidentiality of p is written $p^{\rightarrow}$, and intuitively represents the authority to observe secrets owned by p. The integrity of p, written $p^{\leftarrow}$, represents the authority to affect information owned by p. A second ordering on principals, defines permitted information flows. The statement p flows to q, written $p \sqsubseteq q$, denotes that information is permitted to flow from p to q. The ordering $\sqsubseteq$ forms another lattice over principals, which is orthogonal to the authority lattice. The meet and join in the information flow order are written $\sqcap$ and $\sqcup$. Any FLAM principal can be represented as a conjunction of a confidentiality projection and integrity projection $p^{\rightarrow} \wedge q^{\leftarrow}$. Labels of this form are said to be in normal form.

$$
\begin{aligned}
\mathcal{B}[\![p]\!] &\triangleq (b_p, b_p) \\
\mathcal{B}[\![p^{\rightarrow}]\!] &\triangleq (\mathcal{B}_c[\![p]\!], b_{max}) \\
\mathcal{B}[\![p^{\leftarrow}]\!] &\triangleq (b_{min}, \mathcal{B}_i[\![p]\!]) \\
\mathcal{B}[\![p \wedge q]\!] &\triangleq (\max_b\{\mathcal{B}_c[\![p]\!], \mathcal{B}_c[\![q]\!]\}, \max_b\{\mathcal{B}_i[\![p]\!], \mathcal{B}_i[\![q]\!]\}) \\
\mathcal{B}[\![p \vee q]\!] &\triangleq (\min_b\{\mathcal{B}_c[\![p]\!], \mathcal{B}_c[\![q]\!]\}, \min_b\{\mathcal{B}_i[\![p]\!], \mathcal{B}_i[\![q]\!]\}) \\
\mathcal{B}[\![p \sqcup q]\!] &\triangleq (\max_b\{\mathcal{B}_c[\![p]\!], \mathcal{B}_c[\![q]\!]\}, \min_b\{\mathcal{B}_i[\![p]\!], \mathcal{B}_i[\![q]\!]\}) \\
\mathcal{B}[\![p \sqcap q]\!] &\triangleq (\min_b\{\mathcal{B}_c[\![p]\!], \mathcal{B}_c[\![q]\!]\}, \max_b\{\mathcal{B}_i[\![p]\!], \mathcal{B}_i[\![q]\!]\})
\end{aligned}
$$

Figure 4.3: Representing FLAM labels with hypercube labels.

Mapping FLAM Labels to Hypercube Labels

FLAM labels are easily represented in the hypercube model using bit vectors. FLAM labels in normal form map directly to confidentiality and integrity components of hypercube labels. Primitive principals p are mapped to numeric constants, b_p. For example, if there are four 1-bit dimensions and p and q are mutually distrusting, one might map p to `1000` and q to `0100`. Figure 4.3 shows how compound principals can be mapped to hypercube labels. Here, $\mathcal{B}[\![p]\!]$ denotes the representation of p as a pair of its hypercube label components. The confidentiality component is the first in the pair, and the integrity component is the second. $\mathcal{B}_c[\![p]\!]$ denotes the confidentiality component of p and $\mathcal{B}_i[\![p]\!]$ is the integrity component. The values b_{min} and b_{max} are the lowest and greatest bit vectors that can be represented with the width of a label; they are respectively a sequence of all $1s$ and a sequence of all $0s$. Here, $\max_b$ is a function that computes the dimension-wise maximum of two hypercube labels, and $\min_b$ similarly computes a minimum. For example, if there are three, four-bit dimensions, then $mboxmax_b(0x203, 0x302) = 0x303$ and $mboxmin_b(0x202, 0x202) = 0x202$.

```
# Set Page Labels. cur_lvl: {\bot-> & \top <-}
li x1, 0x84      # {B-> & P<-}
li x2, 0x48      # {P-> & B<-}
la x3, prep_to_bob
la x4, bob_to_prep
setmem x1, 0(x3)
setmem x2, 0(x4)

...

# Bob Sends. cur_lvl: {B}
la x5, bob_to_prep
swdwn x6, 0(x5) #decl {B} to {P-> & B<-}

...

# TP Receives. cur_lvl: {(B&P)-> & P<-}
la x5, bob_to_prep
lwdwn x6, 0(x5) #endo {P-> & B<-} to {(B&P)-> & P<-}
```

Figure 4.4: IPC Example.

Interprocess Communication

Figure 4.4 shows an example of how messages are communicated among processes in the tax-preparer application, and more generally, shows how shared memory IPC works in HyperFlow. In the example, $0x80$ represents the principal Bob (B), and $0x04$ represents the Tax Preparer principal (P). A page of memory is allocated for Bob to send messages to Preparer with label $B^{\rightarrow} \wedge P^{\leftarrow}$, and for Preparer to send messages to Bob with label $P^{\rightarrow} \wedge B^{\leftarrow}$. Public and trusted code initializes the labels of the pages used for IPC. In the code segment shown, Bob has a ℓ_{cur} label of B. Because the Tax Preparer is an instance of the tax preparation service specifically for handling Bob's requests, it has a ℓ_{cur} label of $(B \wedge P)^{\rightarrow} \wedge P^{\leftarrow}$ so that it can see Bob's data. For Bob to send a message to Preparer, it simply performs a `SWDWN` instruction on a register with label B which downgrades the register contents to the label of the destination page ($P^{\rightarrow} \wedge B^{\leftarrow}$). This downgrade is robust because Bob has sufficient integrity to remove the $B^{\rightarrow}$

component of the label. The $P^{\rightarrow}$ component of the label can be added because this increases the restrictiveness of the label. The Preparer receives the message by doing a `LWDWN` instruction which endorses the integrity of the message to $P^{\leftarrow}$.

In some cases, it is possible to receive a message through IPC by first downgrading a register and then doing a conventional load instruction to the downgraded register. However, this example demonstrates that this is not always possible, and that the `LWDWN` instruction is necessary for expressiveness of the ISA. The tax preparer cannot endorse the integrity of a register to $B^{\leftarrow}$ because its label does not flow to $B^{\leftarrow}$. However, it can endorse the $P^{\rightarrow} \wedge B^{\leftarrow}$ data to $P^{\leftarrow}$, so it can receive the data with a `LWDWN` instruction.

In this example, we used two separate pages for communication in each direction. However, it is more conventional for processes to share a single page that both processes can both read and write. The label model of HyperFlow is also expressive enough to support bi-directionally shared pages—a single page could be labeled $B \vee P$. With this label, both B and P can write to the page, and both process can read from the page by endorsing it. However, with the aforementioned numerical representations of B and P, $B \vee P$ computes to the fully public and fully distrusted label, which any other process can read and write as well. By representing B as $0b01001$ and P as $0b00101$, $B \vee P$ becomes the more restrictive label, $0b00001$. Therefore, by adding an extra bit, we can represent disjunctions in a way that is distinct from the bottom label. Because this has a one-bit overhead in the size of labels, we chose to use the more compact label encoding and use two pages for communication. Other representations are also possible. For example, using 2-dimensional, rather than 4-dimensional hypercubes offers a compact encoding while retaining unique disjunctions, however,

```
# Gate Registration: cur_lvl {\bot-> & \top<-}
la x1,     switch_process
li x2,     0x0F # {\bot-> & \top<-}
reglcall x1, x2

# Bob. cur_lvl: {B-> & B<-}
...         # compute form, store in shared page
li x4, 0x0
la x3, swith_process
declreg x1, x4 # Flag to choose Bob or Preparer
declreg x2, x4 # Address to jump to after call
lcallr 0(x3)

# Process Switch Call: cur_lvl {\bot-> & \top<-}
switch_process:
li x4, 0xF
endoreg x1, x4 # Flag to choose Bob or Preparer
endoreg x2, x4 # Address to jump to after call
...          # Set labels, jump to target
```

Figure 4.5: System call example.

it reduces the total number of physical principals.

System Calls

In the tax preparer, a single trusted system call is used to manage Bob and the Preparer. This system call starts a new process by initializing the labels for the process and then jumping to the entry point of the process. This system call is implemented as a call gate. Figure 4.5 shows a small segment of the call gate as well as how the gate is registered and called, and by extension, the example shows how system calls can be implemented in HyperFlow more generally. Initially, fully trusted and fully public code registers the call gate at address `switch_process` and with label $\bot^{\rightarrow} \wedge \top^{\leftarrow}$. The call gate takes two arguments. One describes whether labels should be initialized for Bob or the Preparer, and the other is the PC value that is the entry point for the next process. When Bob is done computing, it executes the call gate. Before doing so, it must declassify

the confidentiality of the two arguments from $B^{\rightarrow}$ to $\bot^{\rightarrow}$. It then simply calls the gate with an `LCALLR` instruction.

At the start of the call gate, the handler must endorse the integrity of the two arguments from $B^{\leftarrow}$ to $\top^{\leftarrow}$ because the system call handler is fully trusted, and it takes a conditional branch based on the value of the argument. The call gate handler then sets the tags of all registers and the levels of ℓ_{nlwr}, ℓ_{ncur}, and ℓ_{nupr} to values that depend on the next principal to execute. It then does a `SETBOUNDS` instruction before jumping to the entry point of the next process.

In conventional processors, system calls work by first jumping to trusted code that contains a system call handler table – the particular call handler to execute is selected by using a register argument that contains the call number. This model is also supported by HyperFlow. However, because HyperFlow replaces conventional privilege modes with lattice model information flow labels, the call gates of HyperFlow are more general and can both improve performance and the precision with which access controls are enforced.

Tax Preparation Application

To test the usability of HyperFlow, we implemented the tax-preparer application in assembly using the HyperFlow ISA. Bob has ℓ_{lwr} and ℓ_{upr} labels that are $B^{\leftarrow}$ and $B^{\rightarrow}$ respectively, and generally operates with a ℓ_{cur} label of B. The tax-preparer generally operates with ℓ_{cur} of $(B \wedge P)^{\rightarrow} \wedge P^{\leftarrow}$ because it is an instance of the tax preparation service specifically for handling Bob's requests, so it needs to be able to observe Bob's data. Its ℓ_{lwr} and ℓ_{upr} labels are $P^{\leftarrow}$ and $(B \wedge P)^{\rightarrow}$ respectively.

Before either Bob or the tax-preparer executes, a label manager that is fully trusted and public registers the `switch_process` call gate and initializes the memory label. Bob computes his tax form and sends the message to the preparer using shared memory as described in Section 4.4.7. Bob then yields the processor by calling the `switch_process` gate so that the preparer can begin executing. The tax preparer receives the message and then computes the form using its proprietary data before declassifying the result. The preparer sends the result back to Bob via IPC and yields the processor back to Bob by calling the `switch_process` gate again. Finally, Bob receives the computed form.

4.5 Discussion

In this section we discuss some of the design and implementation trade-offs we made in order to satisfy the goal of making our HyperFlow prototype label-check with ChiselFlow. This section also discusses the process of implementing a processor that label-checks and how this process differs from conventional hardware design without label-checking.

Labels of Labels In ChiselFlow, wires can be used to represent information flow labels that can change at run-time. Because these wires are still wires, they are also labeled. In most information flow systems, it is assumed that information flow labels are fully trusted and fully public. If the labels cannot be trusted, then they clearly cannot be used to establish security, and if they contain secrets, even inspecting them to implement access controls might leak those secrets.

However, because HyperFlow is timing-channel free and implements fine-grained per-register and per-page data labels, we found it necessary to give

more restrictive labels to some of the signals that represent data labels. For example, the per-page data labels are stored in the cache and used as security types for the data in the cache. Most control signals in the cache are labeled with ℓ_{cur} because they represent signals that affect timing. Because the values of these control signals influence the time that per-page data labels are brought into the cache, the labels themselves must be labeled with ℓ_{cur}.

Register File Labeling The register file tags underwent several revisions even though it is a simple component. This is related to the issue of labeling labels because the register file tags are the labels of the registers. Initially, we labeled the register file tags with ℓ_{cur} following the same design choice that we made for the data labels in the cache. However, this would have required the register file tags to be cleared whenever ℓ_{cur} moves downward in the lattice, and by extension the associated registers would need to be cleared. Clearing the registers whenever ℓ_{cur} was lowered would cause unacceptable limitations in the expressiveness of the ISA.

We later tried labeling the registers with ℓ_{lwr}. With this labeling, the level of ℓ_{cur} can move freely without clearing the tags because $\ell_{cur} \sqsubseteq \ell_{lwr}$ at all times. The tags (and registers) need only be cleared when ℓ_{lwr} is changed—effectively, whenever the active process is changed. However, the tags could only be set when $\ell_{cur} = \ell_{lwr}$. This design point is plausibly acceptable, but it still imposed significant restrictions. For example, it is useful to set register tags while at a level above ℓ_{lwr} so that a spare register does not have to retain the value of ℓ_{lwr}. It is also potentially useful to avoid clearing the tags even when ℓ_{lwr} changes, for example during system calls.

In our final design, the register file tags (though not the general purpose

registers themselves) are labeled totally public and totally trusted. As we discuss in 4.4.3, the control signals from the instruction decode unit that determine when tag-setting instructions happen are downgraded. Given that these downgrades only happen during particular instructions and that the information released is clear, these downgrades do not violate our design goal of making information release ISA-visible.

Automatic Tag Propagation Initially, we expected that security labels could be propagated automatically. For example, following an `ADD RS3, RS2, RS1` instruction, we would like to compute the join of the labels of `RS1` and `RS2` and store the result in `RS3` without needing explicit instructions to set the tag of `RS3`. In fact, this form of dynamic tag propagation is common in tagged hardware architectures [20, 82]. However, whether or not general purpose registers are updated depends on control signals that are influenced by timing and labeled with ℓ_{cur}, but the labels of the labels of registers are public and trusted. In other words, dynamically updating the security tags themselves introduces subtle timing channels. We found it preferable to only allow register tags to be updated by explicit instructions that are controlled by the software. Tag updates can be done automatically by the compiler.

Multi-Cycle Execute Unit Stall Signals Our HyperFlow prototype has two execute units that take multiple cycles to perform a computation, the multiplier and the FPU. The time to complete these computations depends on data values. Because the data values have security labels that might not flow to ℓ_{cur}, the time to finish these computations could create timing channel vulnerabilities if they are not carefully controlled. In the HDL code, this timing channel is visible as a flow from the operands from the register file, which have labels that depend

on their tags, to the stall signal, which has the label ℓ_{cur}. We address this timing channel for each of the two execute units in different ways. For the FPU, we do not permit computations on operands with labels that do not flow to ℓ_{cur}. For the multiplier, we permit computations on operands that do not flow to ℓ_{cur}, but the operations always complete in the worst-case time. This presents a trade-off between the expressiveness of the ISA and performance. We chose to take two different approaches for each primarily to demonstrate that either can be statically checked with the information flow type system.

For the FPU, when the labels of the operands do not flow to ℓ_{cur}, the stall signals in the pipeline are modified to hide the stall signal that is an output from the FPU – this dynamic check converts the insecure information flow to a correctness error if the arguments to the FPU ever have operands with labels that do not flow to ℓ_{cur}. Access controls guard the inputs to the FPU to ensure that this invariant always holds, and a correctness violation is never introduced in practice.

The multiplier is modified so that it always takes in the maximum number of cycles.In this way it is always correct, but does not leak information through timing. Unlike the FPU, multiplications can be performed on operands with labels that do not flow to ℓ_{cur}. However, implementing a constant-time execute unit is not straightforward. Because the multiplier is a simple FSM, it is a better candidate for this approach than the FPU, which can be viewed as an independent coprocessor. To implement a constant-time multiplier, we used two state registers: the primary state register and the shadow state register. The primary state register always points to the FSM state farthest from the terminal state. The shadow state is always updated based on the control signals and operands

to point to what the state *would have been* in the original multiplier. The primary state does not depend on the operands, and so its label only depends on the label ℓ_{cur}. The label of the shadow state does depend on the label of the operands, and the value of the shadow state is used to compute the output, but the stall signal does not depend on the shadow state. In this way the multiplier label-checks and is correct.

It is also possible to implement a multiplier that supports both constant-time multiplications of values that cannot be read by ℓ_{cur} and faster multiplications of values with labels that do flow to ℓ_{cur}. The multiplier could compare the labels before multiplication begins. When the labels of the arguments do flow to ℓ_{cur}, the shadow state can be used to generate the stall signal rather than the primary state. This design would incur just a small amount of area overhead compared to the constant-time multiplier; it would require an additional MUX to decide which state register to use, and an additional label comparison would be needed.

HyperFlow and the Spectre and Meltdown Attacks The recent Spectre [42] and Meltdown [52] attacks exploit out-of-order and speculative execution. HyperFlow, like the RocketChip baseline it extends, does not support out-of-order execution or speculative execution. It does, however, speculatively update the branch history table (BHT) and speculatively update the instruction cache. The BHT in RocketChip includes a table of 2-bit counters per index and a global history table. On a hit in the branch-target buffer (BTB), the BHT is updated speculatively. Although the BHT and BTB are accessed in the fetch stage of the processor pipeline, the speculative update is not undone until the memory stage by reseting the global history table on branch miss-predictions. In an unsafe de-

sign, it might be possible for a secret instruction to cause a speculative update to the BHT that is visible to a public instruction earlier in the pipeline, leaking information through timing. In HyperFlow, however, when ℓ_{cur} moves downward in the lattice, the BTB is invalidated and cleared, and both the BHT global history register and branch table are cleared in the same cycle that ℓ_{cur} changes.

The Meltdown attack exploits a vulnerability in memory permissions checks in the data caches of Intel processors [52]. In some Intel processors, the data fetch and TLB permissions checks that a memory access entails are implemented with separate micro-ops. Because the data access can happen before the permissions check, it is possible that the data access can modify the cache even though the TLB permissions check later rejects the access. The speculatively modified cache state creates a timing channel. In HyperFlow, a similar potential vulnerability is prevented in two ways. First, the cache implementation is guarded by a timing label that represents the secrecy of the process that brought the entries into the cache. When ℓ_{cur} becomes lower than HyperFlow, the cache is invalidated. Second, permissions checks that govern memory data are tightly coupled with data access. The memory tags governing the accessed data are inspected in the same cycle of the cache pipeline during which the data is granted.

CHAPTER 5

TIMING COMPARTMENTS: TIMING CHANNEL PROTECTION FOR A MULTI-CORE PROCESSOR

HyperFlow addresses timing channels among mutually distrusting software modules that share a processor over time through time multiplexing. However, in a multicore processor performance-enhancing features are also shared among concurrently executing processes, causing additional timing channels. This chapter presents Timing Compartments, a hardware architecture that prevents timing channels among concurrently executing processes in a multicore processor.

At a high level, the timing compartments architecture works by exposing the process ID of the software executing on each core. The hardware then prevents timing interference among processes by temporally and spatially partitioning resources among processes identified by this ID. Because spatial and temporal partitioning enforce strict noninterference, Timing Compartments is amenable to static information flow control analysis by an HDL type system. This work also identifies and addresses new timing channels not found in prior studies including timing channels caused by contention for shared MSHR queues and contention among cache coherence transactions originating from different security domains.

Experimental results suggest that straightforwardly applying temporal and spatial partitioning can have significant performance overheads. We propose two optimization mechanisms that improve performance. First, we propose coordinated scheduling of time slots for time-multiplexed resources in the shared memory hierarchy. Our study shows that coordinated scheduling reduces the

average L2 miss latency by up to 62% compared to an uncoordinated baseline. Second, we propose a novel optimization which increases the available bandwidth through a temporally-partitioned memory controller. As our experiments show that memory bandwidth reduction is the main source of performance overhead, this optimization improves performance considerably. The optimizations reduce the performance overhead of timing compartments by 58%. Simulation results suggest the performance overhead of timing compartments with the proposed optimizations is quite reasonable especially when a small number of compartments run concurrently. Compared to the baseline with no timing isolation, executing two timing compartments reduces system throughput by less than 7% on average and by less than 2% for compute-bound workloads. In the worst case, memory-intensive workloads incur up to an 18% overhead.

5.1 Timing compartments

5.1.1 Objective and scope

The goal of timing compartments is to provide timing isolation among software running concurrently on a multi-core processor. Ideally, timing compartments should provide similar security as running the same software on dedicated processors. Therefore, the focus is on removing timing interference among parallel processing cores.

Timing compartments ensure that the timing of a program in one compartment is independent of program behavior in other compartments. In doing so, timing compartments prevents both covert timing channels that are intentional, and unintentional covert channels (i.e., side channels). However, tim-

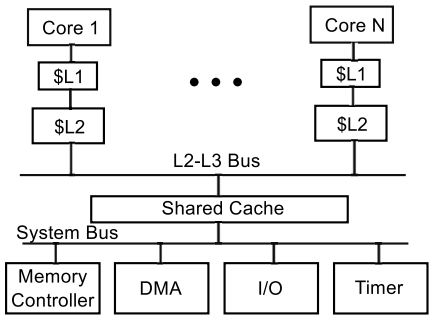

Figure 5.1: Baseline multi-core architecture.

ing compartments do not remove timing dependence within one compartment. For example, Bernstein's attack [5] showed that an AES key in OpenSSL can be extracted by observing timing variations that depend on cache interference among memory accesses within one program. These timing channel vulnerabilities exist even if a program runs on its own dedicated hardware. Since this is an orthogonal problem, they are not prevented by timing compartments. Language-level techniques have been developed to mitigate [108, 3, 107] these timing channels.

5.1.2 Architecture model

Figure 5.1 shows the conventional multi-core architecture that is assumed as the baseline in this chapter. The architecture has multiple cores, each with one or more private caches (L1 and L2). The cores are connected to a shared cache (L3) via an on-chip bus. A shared system bus connects the shared cache to a memory controller that manages requests to main memory as well as other system components such as a DMA engine, a timer, and I/O modules. Bus interconnects are used to model on-chip networks. The general approach and findings should apply to other types of interconnect networks as well.

5.1.3 Threat model and assumptions

Our threat model focuses on software attacks that might enable timing channels among timing compartments. We assume that attackers do not have physical access to the system, and do not consider physical attacks such as ones that tamper with off-chip memory buses or physical side-channel attacks through power consumption or electromagnetic emission. If physical security is required, timing compartments can be combined with existing off-chip memory protection techniques [84, 1, 30].

We also assume that explicit communication between different timing compartments is prevented by allocating disjoint virtual address spaces for software modules in different compartments. Physical addresses should not be shared aside from read-only pages that contain instructions or libraries. There is no point in timing isolation if explicit communication is allowed.

5.2 Protection mechanisms

5.2.1 Approach

Timing channels exist when an adversary can correlate the timing of some event that it can measure with secret-dependent behavior of some other software module. As a result, any program-dependent interference in shared hardware resources between distrusting software modules may lead to timing channels. To achieve a degree of isolation that is comparable to running on separate hardware, timing compartments are designed to remove contention in shared hard-

ware resources.

Timing compartments intend to strictly eliminate timing channels, ideally to enforce noninterference among compartments. Therefore, timing compartments enforce timing channel protection by applying two strict partitioning techniques in micro-archtiectural features that might otherwise cause timing channels. *Spatial partitioning* removes contention by duplicating or partitioning a resource for each compartment. *Temporal partitioning* removes contention by time multiplexing resources among timing compartments with a fixed schedule. By partitioning resources, timing interference among distrusting domains is prevented. Prior work has demonstrated that timing channel vulnerabilities exist when software interferes with the timing of its own events, such as cache accesses [5, 93], or even when there is no interference at all [43]. However, such attacks are present even when software is not co-resident with malicious tenants on the same hardware, and they can be prevented with software-level defenses [3], and pre-loading the cache [43] respectively.

Other timing channel protection mechanisms exist aside from partitioning resources. Defenses can can reduce the ability of the attacker to measure the timing of events, for example, by worsening the clock resolution [60], or disabling performance counters [46]. However, performance counters and precise timing measurements are important for performance tuning and other applications. In the rest of the section, we describe how timing compartments eliminates timing channels in each micro-architectural component, and discuss the trade-off between spatial and temporal partitioning for each component.

5.2.2 Timing compartment ID

To track the timing protection boundaries, management software such as an operating system assigns a timing compartment ID (TCID) to processes. The same TCID can be assigned to multiple processes that do not require strong timing isolation among them such as ones belong to the same user.

In hardware, each processing core has an active timing compartment register (ATC) which indicates the TCID of the TC that is currently executing on that core. The value of the ATC is appended to each memory request and used by enforcement mechanisms to remove interference between different compartments. The size of the ATC is logarithmic with the number of physical cores as hardware only needs to distinguish active compartments running concurrently. The management software can virtualize TCIDs by maintaining a translation between virtual and physical TCIDs.

5.2.3 Private resource protection

To remove timing channels through each core's private resources (such as TLBs, private caches, branch predictors, and pipelines), at most one timing compartment is allocated to a core at a time. Simultaneous multithreading (SMT) is restricted so that only processes with the same TCID can share a core. This approach is already taken in production EC2 servers, which disable SMT to prevent timing channel attacks [111].

However, multiple timing compartments can use these resources through time-sharing. Therefore, there exist timing channels if the state is kept across

Component	Timing Channel	Solution
Shared caches	Replacement	Way partitioning
	MSHRs	Duplicate MSHRs
	Response ports	Separate queues
Memory Controller	DRAM bus	Time multiplexing
	Queueing structure	Separate queues
	Row buffer	Closed Page Policy
	Response ports	Separate queues
On-Chip interconnect	Interconnect bus	Time multiplexing
	Queueing structure	Separate queues
Cache coherence	Coherence bus	Time multiplexing
	Cache port	Response by L3

Table 5.1: Summary of timing channels and protection. Green represents newly identified ones.

context switches. For example, the branch behavior of one timing compartment may affect the next timing compartment if the branch predictor table is kept across a switch. To eliminate this timing channel, timing compartments flushes the per-core state when a core leaves a timing compartment (i.e., the TCID changes).

To prevent information leakage, the time taken to flush should not depend on each timing compartment's state. For example, cache flushing should not take longer when there are more dirty blocks. Therefore, we design hardware to support secure flushing that blocks a core for the worst-case write-back time. In our evaluation, we found the performance impact of the flushing is negligible.

5.2.4 Timing isolation in memory hierarchy

Table 5.1 summarizes the timing channels in the shared memory hierarchy and our approaches to remove them. Newly discovered timing channels are highlighted in green.

Cache contention

Static cache partitioning [70] eliminates cache interference among timing compartments by allowing a cache block to replace only entries owned by the same timing compartment. While other approaches to cache protection have been proposed [93], timing compartments use way partitioning because partitioning completely eliminates timing channels. Way partitioning is a form of spatial partitioning that allocates each cache way to one timing compartment. Cache partition control registers (CPCs) associate a TCID with each way. On a cache access, only entries in ways owned by the corresponding TC are checked or evicted. By changing these registers, management software can adjust the number of ways allocated to each TC. As with private caches, any partitions owned by a TC must be flushed when it is context switched out.

For shared caches, spatial partitioning provides better performance than temporal partitioning by allowing all active TCs to use a portion of the cache and keep the most heavily used data on-chip. In temporal partitioning, while one TC can use the entire cache, the performance of other TCs will significantly degrade as their memory accesses need to go off-chip.

MSHR Contention. In addition to contention for cache arrays, MSHRs also require protection. Contention for miss status holding registers (MSHRs) in non-blocking caches causes timing channels. The number of outstanding misses that the cache can tolerate depends on the number of MSHRs. Once all MSHRs are occupied, the cache will stall on a miss, resulting in increased latency for cache accesses. Therefore, shared MSHRs cause a timing channel. To remove MSHR contention, disjoint sets of MSHRs are allocated to each timing compartment. MSHR contention is resolved with spatial partitioning instead of tempo-

ral partitioning because MSHRs must be able to serve all active TCs.

Response Port Contention. Conventional caches have CPU-side ports and memory-side ports which are each split into request and response ports. However, each port can only service a single response/request at a time, creating timing channels through contention. Similarly, shared queues that buffer responses at the ports also lead to timing interference. To remove this timing channel, the cache ports are time multiplexed and the shared queue is partitioned into per-compartment queues. Temporal partitioning is strictly better than spatial partitioning here. The ports are only interfaced with temporally partitioned networks, so there is no benefit from duplication.

On-Chip interconnect contention

For on-chip networks, we adopt a static time multiplexing approach proposed in previous work [94], and extend it with the capability to allow system software to control network bandwidth allocation and scheduling. Each network arbiter is extended with a ring buffer of network turn control (NTC) registers and a network turn offset control (NTOC) register. NTCs specify a TCID and turn length. The NTOC allows the start of the bus schedule to be adjusted relative to other time multiplexed resources (namely, other buses and the memory controller). Temporal partitioning is preferable to spatial partitioning because bus transactions are short, and duplicating the network would have high area overhead.

Main memory controller contention

The main memory is shared concurrently by multiple cores. As a result, interference among memory accesses from multiple TCs leads to timing channels. We temporally partition the memory controller [89] to provide memory protection, but propose a new optimization to reduce its overhead (Section 5.3.2).

This approach uses a set of techniques to remove timing channels in each memory controller component. A shared request queue is replaced with smaller per-compartment queues. To remove timing variation based on row buffer state, the DRAM controller uses a closed page policy. A closed page clears the row buffers by pre-charging them after each row access. This is effectively flushing at the end of a temporal partitioning turn. Contention for DRAM resources such as the command/data bus, banks, and ranks is removed with temporal partitioning. A period during which no new requests can be issued, called the *dead time*, is added to each time slice in order to prevent in-flight requests or refreshes from interfering with the next time slice. A hybrid temporal-spatial partitioning approach can also be taken. In a system with multiple memory channels, a subset of the TCs can be assigned to each channel, and temporal partitioning can be used within each channel.

Memory controller protection supports fine-grained resource allocation by the OS. The resource allocation control structures are similar to those used for the on-chip interconnects. A ring buffer of memory turn control registers (MCTs) controls the owner and length of each turn and the memory turn offset control register (MTOC) controls the turn offset.

Contention in cache coherence protocols

We found that cache coherence protocols can be a source of timing channels. Coherence operations can lead to timing interference though coherence bus contention or contention for cache ports. Even when there is no shared data between timing compartments, traffic on the snooping coherence bus can lead to a timing channel because the bus is shared by multiple TCs.

Attack Example. Here, we demonstrate a timing, covert-channel attack through cache coherence mechanisms using a simulated 4-core system. Each core has private L1 and L2 caches, and the four cores share an L3 cache. The four L2 caches are connected with a snooping coherence bus which uses a MOESI protocol. TC0 runs on core 0 and core 1 while TC1 runs on core 2 and core 3.

TC0 runs two threads on different cores. Both threads run a `for` loop, and writes to shared data during each iteration. Before each write is performed, one of the L2 caches has to forward the data to the other through the snooping coherence bus and invalidate its own copy. TC0 repeats this process and records the time for each loop. To communicate a secret, TC1 sends a '0' by doing nothing and sends a '1' by spawning multiple threads that write to shared data. Figure 5.2 shows the execution time of the `for` loop that TC0 observes, which shows clear correlation to the secret. '01101100', sent by TC1.

Protection. Cache coherence mechanisms have two sources of timing interference: bus contention and port contention. As with the on-chip data bus, we eliminate interference with temporal partitioning. However, timing channel protection for the coherence mechanism is different from data bus protection in two ways. While coherence requests are associated with the TCID of the

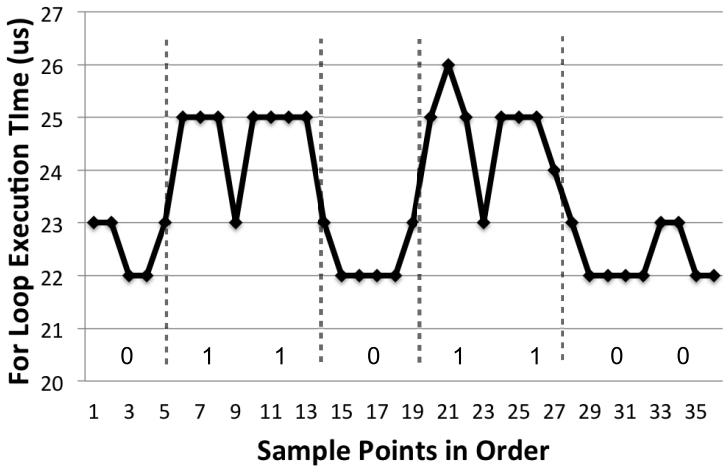

Figure 5.2: TC0's timing observation.

core that issues the request, responses must be tagged with the TCID of the corresponding request, not the TCID of the core that sends the response. In the MOESI coherence protocol, a private cache that owns the data may need to send it to another cache. These transactions can contend for cache ports with requests from processing pipelines. To remove this contention, we change the coherence protocol to serve data from the shared cache instead of from the private caches whenever the data is owned by a different timing compartment. Because protected pages that are shared between compartments are always read-only, the shared cache or memory always has an up-to-date copy.

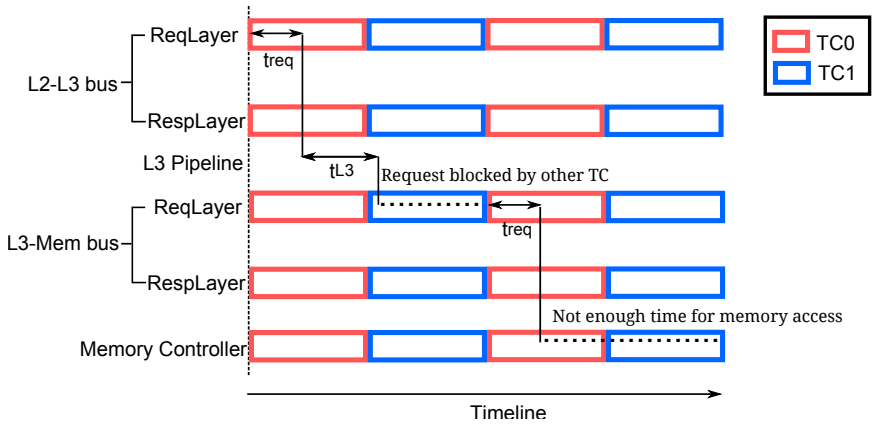

Figure 5.3: A bad time multiplexing schedule.

5.3 Performance optimizations

5.3.1 Time-slice coordination

Timing compartments rely heavily on time multiplexing to protect shared resources including the L2-L3 bus, the L3-memory bus, and the memory controller. Since these resources are all involved in handling L2 misses, their schedules must be coordinated to achieve high performance. For example, to avoid an unnecessary delay when a request exits the L3-memory bus, the memory controller should be available immediately to handle that request.

Figure 5.3 illustrates the problem. It shows when each of two timing compartments are scheduled to use the time multiplexed resources along the L2 miss path. Red (Blue) blocks indicate that TC0 (TC1) is scheduled to use the device. In the figure, an access from TC0 that misses in both the L2 and L3 caches is shown. The L2 miss sends a request to the L3 using the L2-L3 bus request layer. When the L3 access is complete, the request must proceed through the L3-memory bus request layer, but at this time TC1 is scheduled to use the L3-

memory bus, so TC0 is blocked until TC1 finishes. Then, when it arrives at the memory controller, there is not enough time left to complete a request, so it is blocked again.

The time multiplexing schedule should be coordinated among related resources to avoid this problem. We define a *turn* as the block of time that a TC is scheduled to use a resource, and a *turn length* is the duration of a turn. An *offset* refers to a shift in the start of the turn for a single resource compared to the start of the full schedule. Coordination can be done by controlling the turn lengths and offsets for each time multiplexed resource.

There are three main criteria for developing an efficient schedule. First, the turn length should be long enough for at least one transaction to complete. Second, to reduce unnecessary delays, the offset should begin each turn when the data is available from the preceding step. Third, the desired schedule should repeat for each timing compartment.

The timing of an L2 miss depends on whether it hits or misses in the L3. After an L3 hit, the response is sent back across the L2-L3 bus immediately. After an L3 miss, the request must propagate through the L3-memory bus, the memory controller, and so on. Since the timing differs for these two cases, they produce conflicting timing constraints.

Given the conflicting requirements, we found that deriving the optimal schedule for memory hierarchy in general is a nonlinear optimization problem which is too difficult to be solved analytically. Instead, we derive a schedule heuristically. We built a custom simulator that models the memory hierarchy components involved in an L2 miss. It accepts a schedule (i.e. turn length and

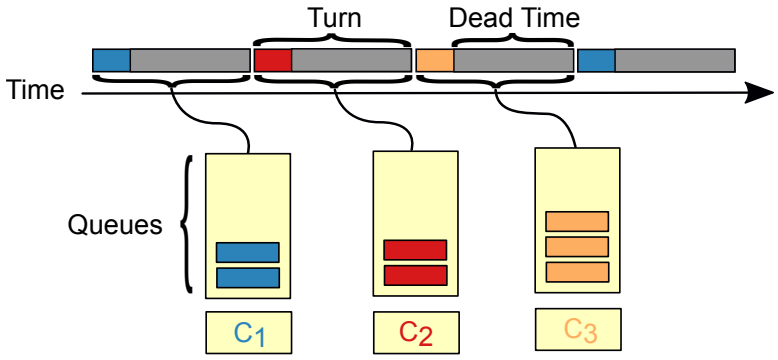

Figure 5.4: A temporal partitioning schedule with three security classes.

offset values) as inputs and calculates the average L2 miss latency assuming the distribution of request arrival times is uniform random. We then used a simulated annealing optimizer to find a schedule that minimizes the L2 miss latency assuming an L3 hit-rate of 90%. We hand-wrote the simulated annealing optimizer in ruby.

The simulation study shows that time-slice coordination has a significant impact on memory latency. For L2 misses that hit in the L3 cache, the worst schedule we found by just varying offset values with a fixed turn length had an average L2 miss latency that is 2.64X higher than the best schedule. For L2 misses in general (90% L2 hit), the schedule found by the optimizer reduced the average L2 miss latency by 62% compared to the worst schedule found, and by 12% compared to a hand-tuned, best-effort schedule.

5.3.2 Operation-aware dead time

Simulations show that the most significant source of overhead for timing compartments is the reduction in maximum usable memory bandwidth due to tem-

porally partitioning the memory controller. Figure 5.4 illustrates temporal partitioning [89] which removes timing interference in a shared memory controller by issuing requests in a fixed, static schedule. Each security domain is allocated to a single time-slice called a turn, and each security domain can only issue transactions during its turn. To prevent transactions issued by one timing compartment from interfering with another timing compartment, the memory controller stops issuing transactions for a period at the end of each turn, called the dead time, in order to ensure that in-flight transactions complete by the end of a turn and do not interfere with the next turn. The dead time is conservatively set to the worst-case time between two transactions. In practice, this is a substantial portion of a time slice. For the parameters used in our simulations, the dead time consumes 22 memory cycles out of each time slice, which range from 23 to 43 cycles.

Security requires that the in-flight transactions from one compartment cannot interfere with transactions issued by another TC in the following turn. However, the worst-case time between two transactions depends on the type of each transaction — in other words, for some transaction types, the worst-case time is lower. Transactions that take less time can safely issue later in the turn. We propose an optimization that leverages this observation by coarsely grouping transactions into reads and writes. Then, a different dead time is used for each type of transaction. For example, the following equations show the worst-case times for each memory operation sequence based on DRAM timing parameters.

- Read, Read: $tFAW - 3 * tRRD$
- Write, Write: $tFAW - 3 * tRRD$
- Read, Write: $tCAS + tBURST + tRTRS - tCWD$

- Write, Read: $tCWD + tBURST + tWTR$

Here, $tCAS$ is the time between a column read command and the placement of data onto the data bus, $tCWD$ is the time between a column write command and the placement of data on the bus, $tBURST$ is the time that the data occupies the data bus, $tRTRS$ is the rank to rank switching time, $tWTR$ is the minimum time between a column write and a column read, $tRRD$ is the minimum time between two row activations, and finally, $tFAW$ is the four-bank activation window, a rolling time period during which no more than for bank activations can occur.

The dead time for reads is set to the worst case time between any read transaction and any other transaction. The dead time for writes is determined similarly. The dead time for each type of transaction determines when that type of transaction can no longer be issued. The dead time for reads is smaller than the dead time for writes, allowing multiple read transactions to be issued in a turn. For example, for the DRAM parameters used in this paper, the dead time for reads is 12 memory cycles whereas the dead time for writes is 18 cycles.

This optimization significantly improves performance for memory-intensive workloads.

5.4 Evaluation

5.4.1 Methodology

To study the performance overhead of timing compartments a timing-protected multicore processor is simulated using gem5 [7] integrated with DRAMSim2 [72]. The simulations use the ARM ISA. Table 5.2 shows the system configuration. The cores use the gem5 "O3" out-of-order core model which runs at 2GHz. Each core has private 32KB L1 instruction and data caches, and a private 256KB L2 cache. The shared L3 cache is varied from 2MB to 9MB depending on the number of cores. The cache configuration parameters are derived from the Intel Xeon E3-1220L and Intel Xeon E7-4820 which are used by Amazon EC2. In DRAMSim2, we simulate a single-channel 667MHz 2GB DDR3 memory. The interconnects in the simulator run at 1GHz.

For most experiments, each core has its own timing compartment. That is, for an n-core system, n timing compartments execute concurrently. We study the impact of having multiple cores in one timing compartment separately. The number of cache ways and the network/memory bandwidths are evenly partitioned among timing compartments. Unless otherwise stated, the memory controller protection uses the minimum turn length (23) and the relaxed dead time optimization, which applies different dead times for reads and writes.

Our experiments use multiprogram workloads, and we describe our methodology precisely enough so that it can be repeated [41]. The simulations are fast-forwarded until each benchmark has executed at least 1 billion instructions. Benchmarks may reach this threshold at different times, meaning

Core count			2/4/6/8
gem5 core model			"O3"
CPU Clock			2GHz
Memory		2GB	667MHz
Network Clock			1GHz
L1d / L1i	32kB	2-way	2 cycles
L2	256kB	8-way	7 cycles
L3	2/4/6/9MB	16-way	17 cycles

Table 5.2: Simulation configuration parameters.

the benchmarks which run faster will be fast-forwarded for more instructions. However, detailed simulations begin from the same point for each workload and for all system configurations. After fast-forwarding, results are collected with a detailed simulation until each core has executed for at least 100M instructions. Statistics are collected for each benchmark at the 100M instruction mark, but all benchmarks continue to run until the simulation ends, so that there is interference for the entire simulation for the insecure baseline.

5.4.2 Performance overhead

This section evaluates the performance overhead of timing isolation by running multiprogram workloads comprised of SPEC2006 benchmarks, and measuring the system throughput (STP). STP is the aggregated normalized IPC of each program relative to the IPC when each program runs by itself. An STP of greater than 1 means that higher throughput is achieved by running the programs in parallel rather than serially. It is computed by

$$\sum_{i=1}^{n} \frac{IPC_{MP,i}}{IPC_{SP,i}}, \tag{5.1}$$

where $IPC_{MP,i}$ is the IPC of the i^{th} program in the workload when run in parallel with the others, and $IPC_{SP,i}$ is the IPC for the same program when it is run

Workload	Benchmarks	Memory Intensity
ast_ast	astar astar	low-low
h26_hm	h264ref hmmer	med-med
ast_h26	astar h264ref	low-med
sjg_h26	sjeng h264ref	med-med
sjg_sjg	sjeng sjeng	med-med
mcf_ast	mcf astar	high-low
lib_ast	libquantum astar	high-low
mcf_mcf	mcf mcf	high-high
mcf_lib	mcf libquantum	high-high
lib_lib	libquantum libquantum	high-high

Table 5.3: Multiprogram workloads.

alone on the same system.

The experiments evaluate the performance for the workload mixes shown in Table 6.2. Workloads were selected to include a diverse set of application mixes that include both memory intensive and compute intensive benchmarks. Applications also vary in cache sensitivity. For experiments with two cores, each workload consists of the two benchmarks in the table. For experiments with more cores, the same labels are used to refer to a workload mix where half the compartments run the first program, and the others run the second half.

Overall performance overhead

Figure 5.5 shows the performance overhead of timing compartments in a system with 2 cores both with and without optimizations. The optimizations include the relaxed dead time for the memory controller (relaxed) and static workload-aware resource allocations for the cache (Cache) and memory (Mem). Workload-aware resource allocation uses the amount of cache space and memory bandwidth allocated to each application to match the general application demands. The allocation, however, is still fixed for the entire execution and thus only reflects general application characteristics, not data-dependent pro-

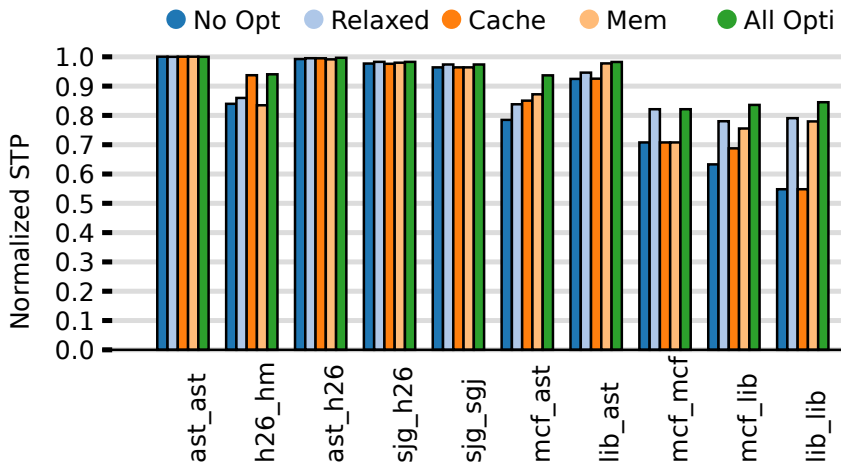

Figure 5.5: Performance Overhead of Timing Compartments.

gram behavior. The bar labeled (All Opti) uses all optimizations. The overhead is measured using the STP of the secure system normalized to the STP of the insecure baseline. Performance overhead depends heavily on memory intensity. The overhead is quite low for compute-intensive workloads such as `sjg_sjg`, `ast_h26`, `ast_ast`, and `sjg_h26`, because timing compartments only incur overhead during L2 misses.

On the other hand, the performance overhead can be quite significant for memory intensive workloads when unoptimized protection techniques are used. For example, `lib_lib` has overhead close to 45%. For these cases, the relaxed operation-aware dead time can significantly reduce overhead. This optimization reduces the worst-case overhead to roughly 20%. The overhead can be further reduced to less than 7% on average and 16% in the worst case if the application-aware resource allocation is also enabled.

We note that 2 TCs are enough to support many practical application scenarios. For example, mobile security platforms such as ARM TrustZone [59] isolate applications into two worlds. Each world can be placed in a single TC regard-

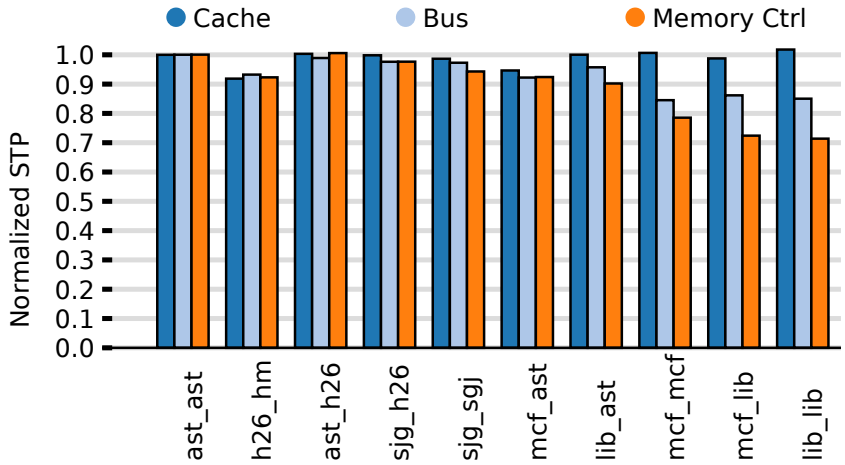

Figure 5.6: Performance Breakdown (4 cores).

less of the number of cores. As we show later, overheads scale with the number of TCs, not with the number of cores. Similarly, hardware compartments such as Intel SGX are designed to be used as a secure co-processor to run security-critical parts of an application and mostly likely to run one compartment at a time.

Overhead breakdown

To better understand the sources of the performance overhead, the overhead of protection mechanisms were evaluated individually. Figure 6.4 shows the performance overhead of timing compartments compared to the insecure baseline when only a single protection mechanism is enabled at a time. These protection mechanisms include cache partitioning, time multiplexing for the on-chip interconnects, and time multiplexing for the memory controller. The memory controller uses the relaxed dead time optimization. The results suggest that the memory controller protection is the most substantial source of overhead, and that cache partitioning and bus protection are less costly. This is because

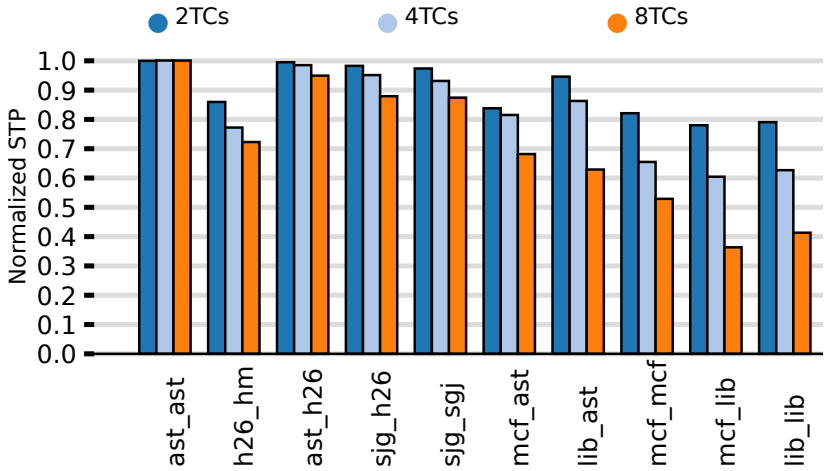

Figure 5.7: Norm. STP of TCs as the number of TCs increases.

memory controller protection requires a dead time [89] to drain in-flight transactions which significantly reduces total memory bandwidth. For example, in our DRAM configuration, the turn length is 23 cycles whereas the dead time is 22 cycles. Prior results using a similar simulation environment, benchmarks, and parameters also suggest that short turn lengths achieve the best performance [89]. As a result, only one DRAM request can be issued every 23 cycles, incurring significant overhead for bandwidth-limited applications. While protection for caches and on-chip interconnects introduces inefficiencies, they do not reduce the total cache capacity or the on-chip interconnect bandwidth.

Scaling the number of TCs

Figure 5.7 shows the performance overhead of timing compartments as the number of TCs and cores increases from 2 to 8. The relaxed dead time optimization is used, but resource allocation is not optimized based on application characteristics. The performance overhead is low (less than 5%) for compute-intensive benchmarks even with a large number of TCs. Yet, the overhead of

memory-intensive workloads increases with the number of TCs, because more compartments share the same amount of fixed memory bandwidth.

The results suggest that many TCs can be supported simultaneously with reasonable overhead for compute-intensive applications. However, many memory intensive workloads should not be allocated to the same machine to keep overheads low. Chapter 6 proposes further optimizations to reduce the overhead of timing channel protection for shared memory controllers.

Using one TC for multiple cores

Multiple programs or virtual machines can be grouped into the same timing compartment as long as they have the same security needs. For example, cloud users often request several VMs that run on the same physical machine, possibly to avoid network communication latencies. Naturally, VMs owned by the same user can be grouped into the same timing compartment. Also, low-security VMs may not need timing channel protection. For multi-threaded programs, one program can also use multiple cores running in the same timing compartment.

Using one timing compartment for multiple cores provides substantial performance improvements because cores within one compartment can share resources as they would in a conventional system without timing protection. Figure 5.8 shows the benefit of grouping multiple programs into a single timing compartment by comparing the STP of a system that runs 4 programs in 4 TCs, to the same system running the same 4 programs in 2TCs. The memory controller turn lengths are increased to 30 for the 2 TC system (compared to 23 for the 4TC systems) since TCs running two programs consume more memory

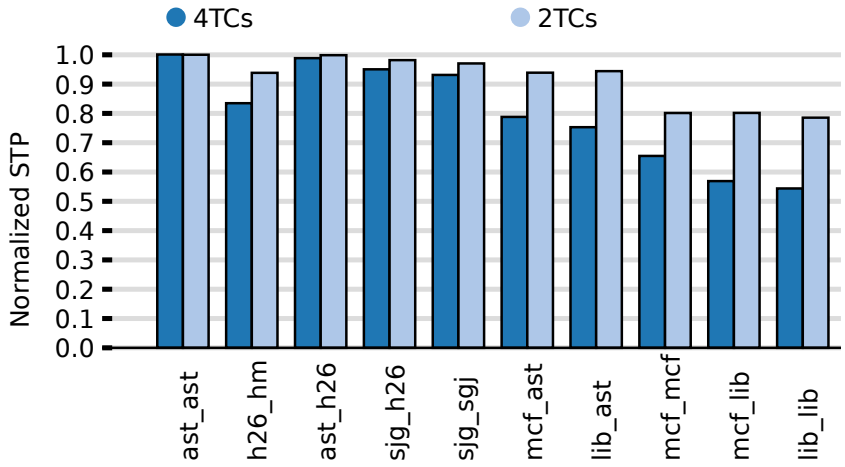

Figure 5.8: Benefit of allowing 2 programs to share a TC.

bandwidth. The performance improvement is dependent on the characteristics of the applications in the workload. For memory intensive workloads such as `lib_lib`, the STP improves by as much as 54% compared to using 4TCs. However, for workloads like `ast_ast` where all applications are compute-bound, the improvement is small. Overall, the performance overhead of running 2 TCs on 4 cores is comparable to running 2 TCs on 2 cores.

Multi-threaded performance overhead

The multiprogram workloads do not show the overhead of protection for the cache coherence bus, because they do not have shared data. To evaluate the overhead of cache coherence protection, we used SPLASH-2 [99] benchmarks on a 4-core system. For each experiment, we run two copies of a SPLASH-2 benchmark, each with two threads, in two TCs.

The overhead of cache coherence protection was evaluated by comparing the normalized execution time of a system with all protection mechanisms to the normalized execution time of the same system with all protection mechanisms

except cache coherence protection. The overhead of adding cache coherence protection is quite low; the overhead is at most 1.5% for `ocean_cp`. The overheads for the remaining SPLASH-2 benchmarks is negligible. The overhead is low because coherence protocol transactions are infrequent.

Context switching overhead

When a timing compartment is context switched out, the remaining state in the private and shared caches, and on-chip resources such as the TLB and branch predictor, need to be flushed and dirty cache lines need to be written back to main memory. To prevent write-back requests from interfering with the incoming process, the core must be stalled until all write-backs are complete. We believe flushing the private and shared caches are the main source of overhead as they are the largest state elements. We evaluated the STP of a 4-core system with 4TCs and with private caches that are flushed every 10ms, 50ms, and 100ms normalized to the STP of the system without flushing. The overhead of context switching is quite small. On average, the overhead is 2.1% when context switches happen every 10ms and 0.8% when context switches happen every 100ms. The overhead is at most 7% (for `h26_hm`) when flushing happens every 10ms.

CHAPTER 6

LATTICE PRIORITY SCHEDULING FOR SHARED MEMORY CONTROLLERS

Chapter 5 identified that in a multicore processor with timing channel protection, memory controller timing channel protection is the performance bottleneck. This chapter proposes a memory controller scheduling algorithm for timing channel protection called lattice priority scheduling (LPS). LPS improves performance by more precisely enforcing the security policies of a system that specifies those policies with lattice model information flow labels, such as the HyperFlow architecture presented in Chapter 4. For example, flows from `public` to `secret` are secure. By allowing these permissible flows, LPS can both improve upon the memory bandwidth that is utilized and dynamically respond to the run-time behavior of applications. We evaluate LPS in a simulated 8-core microprocessor. Compared to a straightforwardly time-multiplexed timing-channel-free memory scheduler, lattice priority scheduling improves system throughput by over 30% on average and by up to 84% for some workloads.

This work is the first to describe an algorithm for allocating a shared, microarchitectural resource among entities while providing timing-channel protection under a security policy expressed in the lattice model. It does so precisely, leveraging permissible flows to improve the efficiency of allocation decisions. The lattice policy is highly expressive, and the proposed algorithms support the full generality of the lattice model.

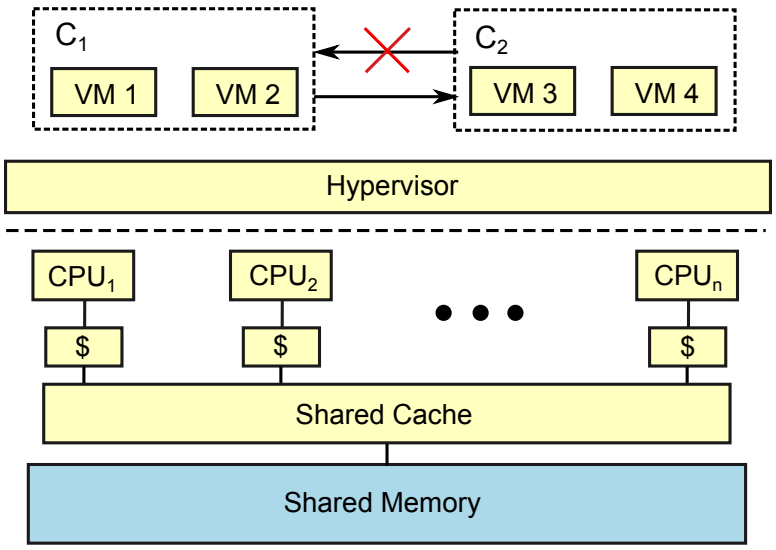

Figure 6.1: System model

6.1 Main-Memory Timing Channels

6.1.1 System Model

This work considers a multi-core processor with two or more cores connected to a shared memory as shown in Figure 6.1. The cores may also share caches, on-chip networks, and other hardware components. The architecture allows processes to be grouped into security classes according to their security needs. In Figure 6.1, the class C_1 does not require timing protection from C_2. However, C_2 distrusts C_1, so timing channels that leak information from C_2 to C_1 must be prevented. As discussed in Section 6.3, lattice priority scheduling supports a wide range of policies describing trust relationships of this form.

This work assumes the target system has a conventional DRAM and memory controller architecture. A modern processor has multiple memory channels, the structure of which is shown in Figure 6.2. Each channel is managed by

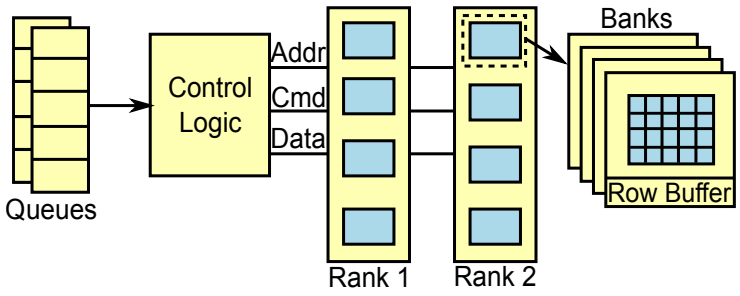

Figure 6.2: A Conventional DRAM Channel

a separate memory controller. Each memory controller has a set of queues of pending memory transactions (read or write requests) and control logic which governs the use of the address, command, and data buses. A DRAM channel is divided into several sets of chips, called ranks, that work in unison to handle each memory transaction. Ranks are further divided into banks. Each bank has an array of DRAM cells which are broken into rows and columns, and each bank has a row buffer that stores the most recently used row. Banks and ranks both improve the parallelism of main memory.

6.1.2 Threat Model

Lattice priority scheduling removes all timing channels that are introduced when a group of co-resident processes share main memory. In particular, this work addresses timing *side channels*, as well as timing *covert channels*. In a side-channel attack, a victim unintentionally leaks a secret to the attacker through timing. For example, Wang et al. [89] present a side-channel attack in which the number of "1s" in a private RSA key is leaked through shared memory traffic. Timing channels enable covert-channel attacks in which one attacker intentionally communicates a secret to another attacker through event timing to bypass a

communication restriction. Wang et al. [89] also present a covert-channel attack where two attackers share a memory. One attacker sends a message by modulating its memory demand. The other attacker issues a large number of memory requests (which interfere with the first attacker's requests), and then measures its memory throughput to receive the message.

LPS addresses a threat model which includes attackers that can run arbitrary programs on the target system and can measure the timing of their own events (e.g., program execution time). The scheduling algorithm is assumed to be public and known to the attacker. Attackers can leverage this information to improve their ability to correlate scheduling decisions with secrets. The threat model includes sophisticated attackers capable of filtering out noise and performing statistical analysis when carrying out both covert and side-channel attacks.

It is assumed that there is adequate protection for explicit communication (such as virtual memory and access controls). The attackers lack physical access to the target system, and therefore cannot execute physical side-channel attacks, such as those which exploit power side channels.

6.1.3 Timing-Channel Attacks in Memory

Conventional memory controllers have timing-channel vulnerabilities due to 1) queue interference, 2) row buffer state, and 3) contention for DRAM resources [89]. In conventional memory controllers, transactions from distrusting processes are placed in a shared queue, where they can interfere, causing measurable delays. Memory banks store the most recently used row in a row buffer for faster access. An attacker can learn that a particular row was used recently if it gets a row buffer hit. Finally, DRAM devices have a number of resources (such as ranks, banks, and the address, command, and data buses) which can service a finite number of simultaneous requests. Contention for these resources also causes timing channels.

6.1.4 Temporal Partitioning

Temporal Partitioning (TP) [89] addresses timing channels in main memory. Fixed Service (FS) [78] improves upon TP, but uses the same high-level approach. This approach prevents all timing leakage between *security classes*, which are groups of processes or virtual machines. Memory transactions are tagged to indicate the security class that owns them. Queue interference is removed by providing separate queues for each security class or statically partitioning a shared queue. The row-buffer timing channel is addressed by using a closed page policy, which pre-charges the row buffer after each read or write command to clear the buffer. The secure memory controller presented in this chapter also uses duplicated/partitioned queues and a closed page policy to address these problems.

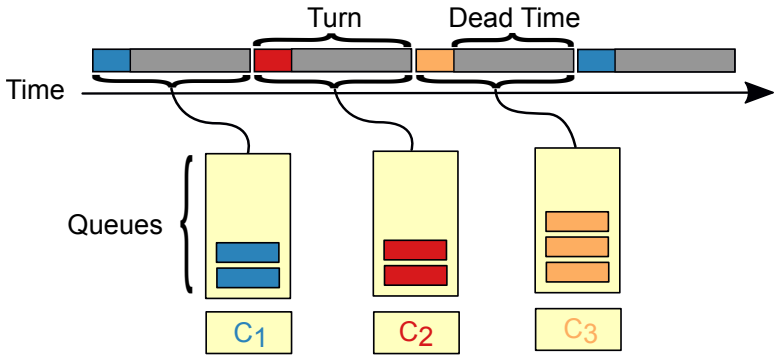

Figure 6.3: A temporal partitioning schedule with three security classes.

TP addresses the contention timing channel through time-division multiplexing. Memory transactions are issued on a fixed, static schedule. Each security class is given a *turn*, which is a time slot in the static schedule during which it is permitted to issue requests, as illustrated in Figure 6.3. Each of the three security classes is allocated a turn in the static schedule. The duration of a turn can be configured to improve performance. The security class currently scheduled with a turn is said to be *active*. If the active security class has no useful work, the turn is wasted. No other security class can use the turn since this would indicate the memory usage of the originally scheduled security class.

Memory transactions require a variable number of cycles to complete. Since the presence of an in-flight transaction from one class could influence the timing of transactions owned by another, any in-flight transactions must be drained before the turn for the next class starts. This is done using a period at the end of the turn called *dead time*, which is long enough to drain the worst-case memory transaction. During dead time, the turn owner can no longer issue transactions. The turn length must be at least as long as dead time. Dead time is indicated in Figure 6.3 by a dark gray segment at the end of each turn.

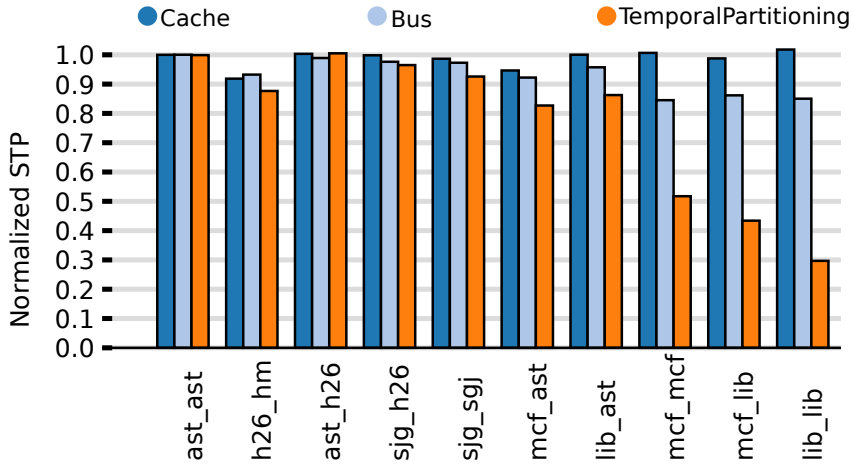

Figure 6.4: System throughput with off-core protection mechanisms normalized to throughput of insecure baseline.

Performance of Temporal Partitioning

Unfortunately, Temporal Partitioning is costly. With 8 cores, TP increases the memory latency by 5.39x compared to the baseline on average, and reduces system throughput (STP) by up to 80%. Additionally, we simulated a 4-core system with timing-channel protection mechanisms applied to all major off-core resources including the shared cache, the interconnects, and the memory controller. The shared caches are statically partitioned, the interconnects are time-multiplexed with a fixed, round-robin schedule, and TP is used to protect the memory controller. Figure 6.4 shows the system throughput with each of these protection mechanisms enabled individually. The throughput is normalized to the system throughput of the same system with all protection mechanisms disabled. Temporal Partitioning is the greatest source of performance overhead and accounts for over 81% of the total overhead of the system. This implies that memory controller protection is the performance bottleneck for systems that remove timing channels through shared off-core hardware resources.

In TP, time multiplexing is the primary source of performance overhead. A transaction can only be issued during the turn of its security class, but not during dead time. Transactions that arrive outside this period must wait in the queue. Therefore, TP increases *queueing delay*: the part of memory latency transaction waits in the queue. The other component of memory latency is *in-flight time*, which is the time from when the transaction issues from the queue until it completes. TP increases queueing delay in several ways.

Dead time reduces the total usable bandwidth compared to a conventional memory controller, increasing queueing delay. With the DRAM timing parameters used in this chapter, each turn must include 43 cycles of dead time. If the minimum turn length is used, transactions are issued at a rate of at most one per 44 memory cycles. Though the turn length can be increased to improve the bandwidth, the bandwidth-latency trade-off favors shorter turns. Increasing the turn length increases the latency imposed on transactions issued by a security class other than the active one. Prior studies [89, 78] confirm that shorter turn lengths achieve the best performance.

In addition to reducing the total usable bandwidth, scheduling constraints restrict TP from efficiently allocating memory bandwidth among security classes. If the active security class has no pending transactions, no other security class can be scheduled in its place as this would leak the demand of the originally scheduled class. Instead, no transactions are issued and memory bandwidth is wasted. Generally, this means that under the security model of TP, the scheduler cannot respond to the dynamic resource needs of each security class leading to performance overhead. We call this problem *demand imprecision*.

Even if bandwidth is allocated to security classes proportionally to the mem-

ory demand from each security class, static scheduling still imposes restrictions which lead to delays. If a transaction from one security class is enqueued when a different security class is active, it is delayed until its security class becomes active.

6.2 Lattice Priority Scheduling

This section proposes a secure scheduling algorithm, called *lattice priority scheduling* (LPS), that enables a timing-safe memory controller to precisely meet the security requirements of the system, thereby improving performance. LPS improves performance by enforcing security policies which include uni-directional protection – in other words, policies which allow information to flow in just one direction between security classes. LPS leverages uni-directional protection to address demand imprecision, remove delays due to static scheduling, and reduce dead time. For simplicity, lattice priority scheduling is first introduced for a system with two security classes, L and H. Information is permitted to flow from L to H, but not from H to L. Section 6.4 generalizes LPS to support arbitrary lattice policies.

Lattice priority scheduling improves upon TP in two ways. First, it schedules security classes dynamically. *Dynamic scheduling* allows LPS to respond to the run-time resource demands of applications, and removes delays caused by static scheduling. Second, it uses *dead time elision*, which improves the total amount of usable memory bandwidth by reducing the dead time.

To illustrate how lattice priority scheduling addresses demand imprecision, Section 6.2.1 proposes a simpler scheduling algorithm based on the concept of

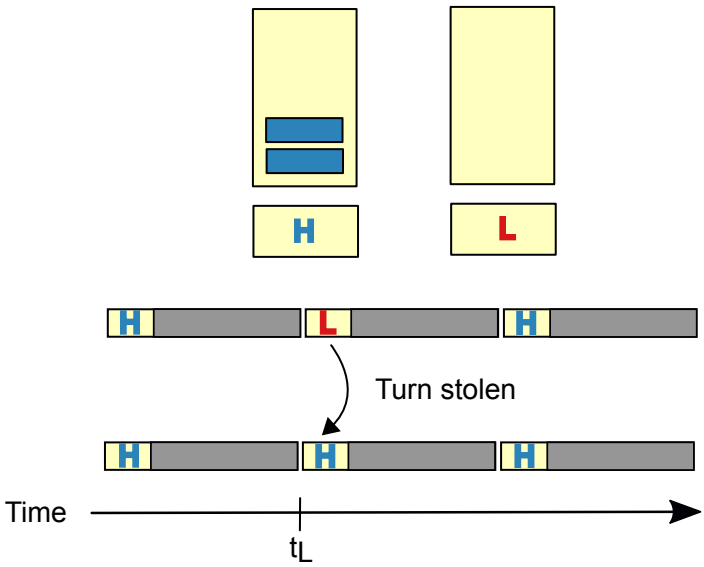

Figure 6.5: Dynamic bandwidth allocation example.

dynamic bandwidth allocation. Then, Section 6.2.2 extends this idea with a fully dynamic schedule, further improving performance.

6.2.1 Dynamic Bandwidth Allocation

Lattice priority scheduling uses dynamic bandwidth allocation to alleviate demand imprecision. To illustrate dynamic bandwidth allocation, we introduce a scheduling algorithm called dynamic bandwidth scheduling (DBS). DBS begins with a static schedule as in TP. Then, at the start of L's turn, DBS checks if L has pending transactions. If it does not, the turn is given to H.

Figure 6.5 shows an example run of DBS. Initially, the TDM schedule is $H, L, \cdots$ The contents of the queues at the start of L's turn at time t_L are shown. Since L has no transactions in its queue at t_L, its turn is reallocated to H which does have pending requests.

The decision to reallocate the turn from L cannot depend on H. If neither have queued transactions at the start of L's turn, L's turn is still given up so that information about H is not leaked to L. The decision to reallocate the turn is final. If L's turn is given up, but it enqueues a transaction later in that turn, L cannot reclaim its turn. If the scheduler allowed turns to be reclaimed, it would leak whether or not the security class which received the bandwidth (H) actually issued a transaction.

Thus, DBS adds a small overhead not present in TP. If neither L nor H have pending transactions at the start of L's turn, it will be given to H. Then, if L enqueues a transaction later in the turn, it cannot be issued. In this case H does not make use of the turn. In TP, L would have kept its turn. However, experiments confirm that this case is exceptional and the overhead is small.

6.2.2 Dynamic Scheduling

The performance of DBS can be improved by scheduling security classes dynamically. The lattice priority scheduling algorithm applies the concept dynamic bandwidth allocation to a fully dynamic schedule, removing delays caused by static scheduling. It is strictly better than DBS. At a high level, lattice priority scheduling prioritizes L over H, and H is scheduled when L has no pending transactions or when L reaches a bandwidth limit.

Since memory transactions take multiple cycles, memory resources must still be granted at the granularity of a turn (including dead time), so that in-flight transactions from different classes do not interfere. Instead of using a static schedule, the arbiter selects which security class is active each turn using Algo-

Algorithm 1 Priority Scheduling

```
1: procedure SELECTTURNOWNER
2:     turn_owner ← ⊥
3:     while ( turn_owner≠ ⊤ and (
           QueueEmpty(turn_owner) or
           not HasBandwidth(turn_owner) ) ) do
4:         turn_owner←AscendFrom(turn_owner)
5:     end while
6:     ConsumeBandwidth(turn_owner)
7:     return turn_owner
8: end procedure
```

rithm 1.

The algorithm searches for the lowest class (`turn_owner`) with pending transactions by checking if the queue of each class is empty. It stops when it finds a security class with pending transactions. It would be insecure for the arbiter to check H and then only schedule L if H did not have pending requests. In this case, L could observe whether or not H had pending transactions. Instead, $L(\bot)$ is the first candidate turn owner. Then, if the candidate turn owner has an empty queue (`QueueEmpty(turn_owner)`), the algorithm calls `AscendFrom(turn_owner)` to select the next security class which can see information from `turn_owner`. In this simple case with only two security classes, `AscendFrom()` always returns H. Section 6.4 describes how `AscendFrom()` works in general. The algorithm stops if there are no other security classes permitted to see information from the candidate. This is true when the candidate is $H(\top)$.

If both L and H are memory-intensive, simply prioritizing L over H would be unfair, and would lead to performance loss for H. Further, L could execute a denial-of-service attack causing H to starve. Instead, priority scheduling places an upper bound on the number of turns that L can be granted within an

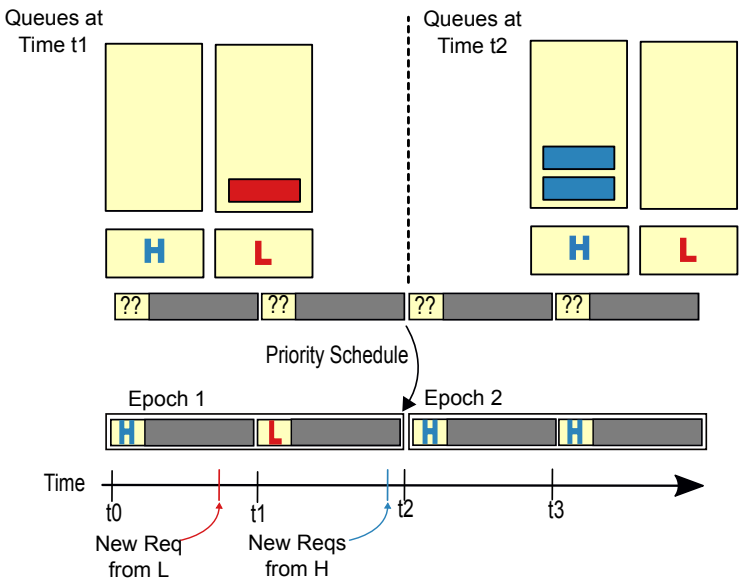

Figure 6.6: Lattice priority example.

epoch, which is a static, fixed-length interval expressed as a number of turns. As Algorithm 1 searches for a turn owner, it calls `HasBandwidth()` which returns true when the argument class has turns left in the epoch. When the turn owner is selected, it calls `ConsumeBandwidth()` to indicate that it has used up a turn. Counters which track the remaining bandwidth for each security class are reset at the start of each epoch. The epoch length and the maximum number of turns granted to each security class can be adjusted depending on static characterizations of the programs, as long as static characterization does not violate security.

Dynamic scheduling with an epoch of 2 turns is similar to turn stealing with the static schedule $L, H, \cdots$, but dynamic scheduling is strictly better. In both cases, H is granted at least one out of every two turns. When L has few requests, H is granted more turns with either approach. However, priority scheduling is more flexible. Figure 6.6 demonstrates the additional flexibility of lattice priority scheduling with an example. Assume the epoch length is 2 and L is

granted at most 1 turn per epoch. The queues for H and L at time t_0 are empty. Since L has no pending requests, H is allocated the turn even though H has no pending requests. Otherwise, L could learn that H did not have a request.

Some time between t_0 and t_1, a new request is enqueued by L. At the start of time t_1, both H and L have pending requests. Since L has higher priority than H, and L has not consumed its maximum of one turn during the first epoch, lattice priority scheduling schedules L. If instead, DBS was used with the static schedule $L, H, \cdots$, the first turn would have been reallocated to H since L had no useful work. However, since the second turn is statically scheduled to H, the turn is not given to L even though H has no pending requests. This is wasteful since L does have pending requests and could make use of the turn.

Continuing with the example run with dynamic scheduling, at time t_2 the second epoch begins and there are two new requests enqueued by H, but none enqueued by L. Both turns are given to H since L has no pending requests. Since this decision depends only on the fact that L has no requests, L learns nothing about whether or not H had any requests. LPS responds to dynamic program behavior just as well as DBS, but can also remove additional delays.

6.2.3 Dead Time Elision

Since H can learn about L, it is permissible for transactions from L to interfere with H. Transactions from L can remain in-flight at the start of H's turn, and the dead time between them can be elided (skipped) as shown in Figure 6.7. By eliding the dead time (shown in gray) during L's turn, L can continue issuing transactions until the turn ends. However, the dead time is still needed at the

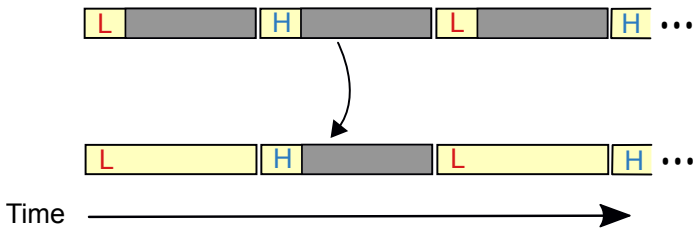

Figure 6.7: Dead-Time Elision.

end of H's turn to prevent transactions from H from interfering with L.

For the system with classes L and H, dead times can be elided whenever L is currently scheduled. The owner of the next turn could be either L or H, but information is permitted to flow from L to either L or H. Section 6.4.2 generalizes dead-time elision to support arbitrary security policies.

6.3 Lattice Security Model

This work leverages the widely accepted lattice model[21] of security to precisely capture the security needs of a wide range of systems. Under the lattice model, entities in a system are assigned a security class. A set of security classes SC, together with an ordering relation $\sqsubseteq$, form a lattice $\langle SC, \sqsubseteq \rangle$. Information is allowed to flow from class $A \in SC$ to $B \in SC$ if and only if $A \sqsubseteq B$ holds. The relation, $\sqsubseteq$, must be reflexive, transitive, and antisymmetric.

Since lattices are a type of *partial* order, not all security classes are necessarily ordered — they may be incomparable. Security classes A and B are *incomparable* if neither $A \sqsubseteq B$ nor $B \sqsubseteq A$ hold. The *meet* of A and B, written $A \sqcap B$, is the greatest class less than both A and B. Similarly, the *join* of A and B, written $A \sqcup B$, is the least class greater than both A and B. For a partial order to be a

lattice, taking the meet or join of any two classes in the lattice must result in a class which is also in the lattice. The classes $\top$ and $\bot$ denote the greatest and least of all classes in the sense that for all classes A, $A \sqsubseteq \top$ and $\bot \sqsubseteq A$ hold. The terms "A is lower than B" and "A is higher than B" are occasionally used as abbreviations for $A \sqsubseteq B$ and $B \sqsubseteq A$ throughout this chapter.

The lattice model is highly expressive, and it can be used to describe the needs of many practical systems. For example, it can be used to describe the Bell-Lapadula multi-level security (MLS) model [45] in which information can flow from `public` to `secret` and from `secret` to `top secret`. Figure 6.8 (a) shows the MLS model in a common pictorial representation of a lattice policy. The arrows show the direction in which information is allowed to flow. For example, an arrow points from `public` to `secret` since `public` $\sqsubseteq$ `secret`. Since the lattice is transitive, it is implied that information can flow from `public` to `top secret` as well. This policy is totally ordered since it contains no incomparable elements.

Incomparable security classes are useful for describing mutual distrust. For example, with the "diamond" lattice shown in Figure 6.8 (b), M_1 and M_2 are incomparable and information cannot flow in either direction between them. However, information from L can flow to M_1 or M_2 and information from both M_1 and M_2 can flow to H. These security classes might be used to describe a cloud system with several low-security VMs (L), two high-security VMs that require timing-channel protection from all other VMs in the system (M_1 and M_2), and a cloud owner class (H) that includes the hypervisor and scheduling/-analysis programs that compute using input data from all the clients.

In this chapter, we assume that a security class is assigned to each process

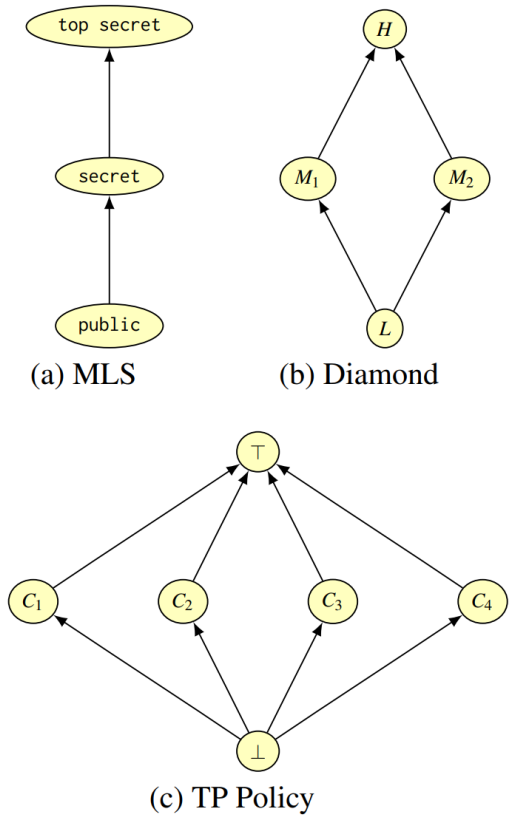

Figure 6.8: Example lattice policies.

(e.g., by the OS or hypervisor). Memory requests are queued and scheduled according to the security class of the process that issued them. The policy determines what scheduling restrictions are needed for protection. The security policy supported by TP can also be described in the lattice model as shown in Figure 6.8 (c) for a system with 4 security classes. Each of the four security classes are incomparable, so the scheduling decisions made by the memory controller must be heavily restricted. The security classes $\top$ and $\bot$ are not actually used, but they are needed formally to represent the join and meet of the other classes. Given this lattice, the approaches described in this chapter will behave equivalently to TP.

6.4 Memory Protection under the Lattice Model

Section 6.2 presents lattice priority scheduling for a system with two security classes, $L \sqsubseteq H$. Practical systems may contain any number of entities, some pairs of which may be mutually distrusting. This section generalizes LPS to support a wider range of systems by leveraging the lattice model

6.4.1 Generalized Dynamic Scheduling

When scheduling security classes, it is preferable to schedule classes that actually have pending transactions. In a memory controller that prevents all timing channels, the time that one class is scheduled cannot depend on the contents of another class's queue. Under a policy in the lattice model, it is permissible for the timing of a security class to depend on the demand from any lower class.

To make the best use of memory bandwidth, LPS searches through the security classes until it finds one that has a pending transaction. A sequence of classes, $C_1, C_2, \cdots, C_i$ that are searched for pending transactions is defined as a *lattice traversal*. An attacker at class C_i that gets scheduled can observe (through timing) that classes $C_1, \cdots, C_{i-1}$ must not have had transactions. Therefore, C_i must be higher than all those checked before it for this to be secure. That is, the sequence in which security classes are searched must form an ascending chain.

As an example, consider three classes, $L \sqsubseteq M \sqsubseteq H$. If the scheduler checks L and finds that it has no transactions, it would be safe to schedule either M or H. If the scheduler then checks M and finds it also has no transactions, it can safely check H. However, if H is checked first, it would be insecure to check M,

since this creates a timing dependence of M on H. Therefore, the only secure traversal that checks all classes is L, M, H.

Since lattices are partial orders, there may be incomparable classes. That is, classes $A, B \in SC$ where neither $A \sqsubseteq B$ nor $B \sqsubseteq A$ is true. Whenever there are incomparable classes, there are multiple ascending chains, and there is no ascending chain which includes all security classes.

For example, in Figure 6.8 (b), both L, M_1, H and L, M_2, H are ascending chains. For fairness it is necessary to ensure that both M_1 and M_2 are scheduled, so both traversals must be used. Though care must be taken — information from M_1 and M_2 cannot be used to decide which ascending chain to use (e.g., it is insecure to simply pick the one with pending transactions). However, both M_1 and M_2 can see information from L since they are both above L. More generally, a class may be chosen from among incomparable classes $C_1, C_2, \cdots C_n$ using information from classes at or below $C_1 \sqcap C_2 \sqcap \cdots \sqcap C_n$.

The lattice priority scheduler simply alternates between incomparable classes in a round-robin fashion. Consider the lattice in Figure 6.8 (b) with two incomparable classes M_1 and M_2 above class L. Assume all classes have turns remaining in this epoch. The first time L has no pending requests, M_1 is granted the turn regardless of whether or not M_1 or M_2 actually have requests. The next time, M_2 is granted the turn, and the time after that M_1 is scheduled, and so on. The decision of which incomparable class to check depends only on how often L has an empty queue, which is acceptable for both M_1 and M_2 to learn.

Using the above intuition, lattice priority scheduling is generalized to support any lattice. `AscendFrom` in Algorithm 1 uses a tree structure as shown in

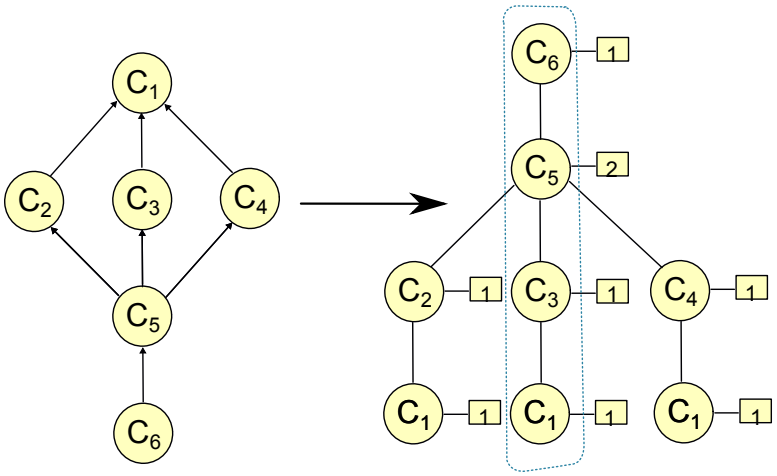

Figure 6.9: Tree structure for selecting traversals in the lattice priority scheduling algorithm.

Figure 6.9 to select an ascending chain. Each node of the tree is a security class. Each parent is directly less than its children (i.e. it is covered by its children). Each node has a counter that increments up to the number of children it has (i.e., the number of classes that it is directly less than in the lattice). Whenever a node is reached during a traversal, its counter is incremented and then used as an index to select from among its children.

In the example shown in Figure 6.9, class C_5 is less than three incomparable classes C_2, C_3, and C_4, so it has a counter that increments from 1 to 3. Assume all classes have bandwidth remaining (turns left) during this epoch. First, C_6 is checked. Since it has no requests, it is not scheduled and C_5 is checked. Since C_5 has no requests either, it is also not scheduled. In this iteration, the counter is 2 so the next class to be offered the turn is C_5's second child, C_3.

Fairness for General Policies

To prevent starvation and improve fairness, security classes are guaranteed a minimum number of turns within each epoch. To enforce this minimum, each class C is given an additional *bandwidth counter* representing the number of turns C can become active this epoch. The counter decreases whenever C uses a turn that it is offered. It is initially the maximum number of turns C can be active in an epoch, which is

$$T_{epoch} - \sum_{C_i \in C+} T_{min}(C_i)$$

where T_{epoch} is the number of turns in an epoch, $C+$ is the set of classes that C is less than, or $\{C_i | C \sqsubset C_i\}$, and $T_{min}(C_i)$ is the minimum number of turns guaranteed to C_i in an epoch.

When selecting the next active class, Algorithm 1 calls `HasBandwidth(turn_owner)` to check if the candidate class, `turn_owner`, has used its maximum number of turns during this epoch. If the bandwidth counter of `turn_owner` is zero, `HasBandwidth(turn_owner)` returns false, preventing `turn_owner` from being scheduled. When `turn_owner` is scheduled, `ConsumeBandwidth(turn_owner)` decrements the bandwidth counter of `turn_owner`.

6.4.2 Generalized Dead Time Elision

In general, dead time can be elided whenever the scheduler is *certain* that the currently scheduled security class will be less than or equal to the next security class to become active (even though the next class may not have been decided yet). The time that the next active security class is decided affects whether or

Algorithm 2 Start-of-Turn Turn Allocation

```
1: procedure AllocateTurn
2:     if IsTurnStart() then
3:         active_class←SelectTurnOwner()
4:     end if
5: end procedure
6: procedure ElideDeadTime
7:     lower_bound ← ⊤
8:     for C ∈ {C′|C′ ⊑ active_class} do
9:         if HasBandwidth(C) then
10:            lower_bound ← lower_bound ⊓ C
11:        end if
12:    end for
13:    return active_class ⊑ lower_bound
14: end procedure
```

not dead time can be elided. There are two suitable choices: 1) at the start of the turn being allocated and 2) at the start of when the dead time would begin if it is not elided.

Algorithm 2 shows the first approach in which the next active security class is decided at the start of the turn. In Algorithm 2, `AllocateTurn` decides which security class is active at the start of each turn by calling `SelectTurnOwner` which is defined in Algorithm 1. Then, `ElideDeadTime` determines if dead time can be elided at the end of this turn. If all the security classes below the currently active one have already consumed all of their bandwidth for this epoch, none of them can become active next turn. In this case, the current active class will always be less than or equal to the next one, so dead time can be skipped. `ElideDeadTime` checks for this condition.

Algorithm 3 shows the second approach, in which the next active class is decided just before dead time will begin if it is needed. It uses `AllocateNext` to pick the next active class just before the start of dead time. Then it uses

Algorithm 3 Dead-Time Turn Allocation

```
1: procedure AllocateTurn
2:     if IsTurnStart() then
3:         active_class←next_active
4:     end if
5: end procedure
6: procedure AllocateNext
7:     if IsStartOfDeadTime() then
8:         next_active← SelectTurnOwner()
9:     end if
10: end procedure
11: procedure ElideDeadTime
12:     return active_class ⊑ next_active
13: end procedure
```

`ElideDeadTime` to decide if dead time can be skipped. Now the class which is scheduled next is known at the time when `ElideDeadTime` is called, so it can simply compare the active class to the next one. At the start of the turn it makes the previously decided next active class (`next_active`) the new active class (`active_class`).

Dead time may be dropped more often when turns are allocated at the start of dead time. However, there is a trade-off. When the turn is allocated sooner, the turn is "locked in" earlier. If in a system with classes $L \sqsubseteq H$, L had no requests at the start of dead time, H would be scheduled next regardless of whether or not H had requests. However, if requests from L arrive later, L cannot reclaim the turn.

6.5 Hardware Implementation

This section describes how the lattice priority scheduling algorithm is implemented in hardware. As with TP, each physical thread (i.e. thread in SMT) has

a register located in the core which stores a security ID representing the security class of the software running in that thread. The memory controller has a number of pending transaction queues equal to the number of threads. Each queue has a register which stores a security ID indicating its owner. The security ID registers in the cores and in the memory controller are managed by the trusted hypervisor or OS. Access controls should prevent modifications by untrusted software.

Unlike TP, lattice priority scheduling supports an arbitrary policy specified in the lattice model. The lattice policy is stored in a dedicated table in the memory controller. The table stores a 1-bit entry for each pair of security classes (A, B) indicating whether or not $A \sqsubseteq B$ is true. The table is initialized and managed by trusted software, and access controls should prevent modifications by untrusted software.

LPS requires registers to configure the number of turns in the epoch and a counter to track the current turn in the epoch. Registers set the maximum bandwidth per epoch for each security class. A set of counters track the bandwidth consumed in the current epoch by each security class.

Unlike TP, which decides the active security class statically, LPS decides the active class for each turn dynamically. Doing so requires checking the queues and bandwidth counters for each security class. This information can be checked for each security class in parallel. A priority encoder is used to select the next turn owner based on the security policy.

While in general a system might have many security classes, the hardware data structures just described need only support as many security classes as

there are physical threads. The security classes can be virtualized to support arbitrarily many classes. The maximum number of simultaneously running security classes is the number of physical threads. Since the number physical threads is small, the area overhead of lattice priority scheduling is small as well. For example, to support a 64 thread system, the policy table would require 512B of storage.

6.6 Evaluation

6.6.1 Methodology

The performance of lattice priority scheduling is evaluated in a multicore out-of-order processor using a simulator based on Gem5[7] integrated with DRAMSim2[72]. Simulation parameters are given in Table 5.2. The experiments simulate 4 cores each with private 32KB L1I/D caches and a private 256kB L2. In our experiments, each core runs at most one security class concurrently. In practice, there may be as many simultaneously executing security classes as there are physical threads. Each core has a private 1MB last-level cache (LLC). Contemporary server processors have a shared LLC with 1MB per thread [39]. These experiments use private caches so that only the direct performance improvement of the memory controller is measured, and changes in cache interference patterns are not measured.

Our experiments use multiprogram workloads, and we describe our methodology precisely enough that it can be repeated [41]. The simulations are fast-forwarded until each benchmark has executed at least 1 billion instructions.

Processor			
Cores and Frequency			4/8cores, 2GHz
Gem5 core model			"O3"
ISA			ARMv8-A
Cache Hierarchy			
L1d / L1i	32kB	2-way	2 cycles
L2 private	256kB	8-way	7 cycles
L3 private	1MB	16-way	10 cycles
Network Clock			1GHz
Memory			
Size and Frequency		8GB	667MHz
Channels, ranks, and banks			1, 8, 8

Table 6.1: Simulator configuration parameters.

Benchmarks may reach this threshold at different times, meaning the benchmarks which run faster will be fast-forwarded for more instructions. However, detailed simulations begin from the same point for each workload and for all system configurations. After fast-forwarding, results are collected with a detailed simulation until each core has executed for at least 100M instructions. Statistics are collected for each benchmark at the 100M instruction mark, but all benchmarks continue to run until the simulation ends, so that there is interference for the entire simulation of the insecure baseline.

The performance evaluation metric is system throughput (STP) which is the aggregate normalized IPC of programs in the multiprogram workload. It is computed as

$$\sum_{i=1}^{n} \frac{IPC_{MP,i}}{IPC_{SP,i}}, \tag{6.1}$$

where $IPC_{MP,i}$ is the IPC of the i^{th} program in the workload when run in parallel with the others, and $IPC_{SP,i}$ is the IPC for the same program when run alone in the same system.

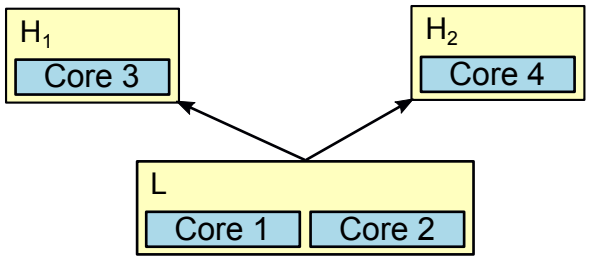

Figure 6.10: Security policy for performance evaluation.

The experiments use multiprogram workloads comprising SPEC benchmarks. Workloads are selected to capture different mixes of memory intensity. Some workloads use the naming convention `bench1_bench2` to indicate that `bench1` is executed on odd-numbered cores and `bench2` is executed on even-numbered cores. Table 6.2 summarizes the remaining workloads by listing benchmarks in order of core number. Note that order matters since LPS schedules security classes differently, and cores may be in different security classes.

The security policy affects the performance of lattice priority scheduling. Unless otherwise stated, these experiments use the policy shown in Figure 6.10, which captures the security requirements of a cloud computing environment with both high and low-confidentiality VMs. Standard VMs run in the security class L, and VMs which require timing-channel protection run in security classes H_1 and H_2. The classes H_1 and H_2 are incomparable and above L. This policy guarantees that information cannot leak out of H_1 or H_2 to any other VM, but relaxes protection for L. Cores 1 and 2 run applications with security class L. Cores 3 and 4 run in security classes H_1 and H_2 respectively. The top security class above H_1 and H_2 (not shown) is not used, and is not allocated any bandwidth.

The turn lengths are chosen based on prior findings that turn lengths that

Workload name	Benchmarks
mix_1	astar x2, libquantum x2
mix_2	astar x3, libquantum
mix_3	h264ref, hmmer, sjeng, libquantum
mix_4	astar x2, mcf x2
mix_5	mcf x2, libquantum x2
mix_6	libquantum x2, hmmer, gobmk
mix_7	libquantum x1, astar x3
mix_8	libquantum x2, mcf x2

Table 6.2: Multiprogram workloads.

are close to the minimum (the dead time) achieve better performance [89]. The dead time is 43 memory cycles. For lattice priority scheduling, the turns are 44 cycles. In all experiments, TP uses three incomparable security classes where Core 1 and 2 share a class. For TP, the same turn length (44) is used for security classes containing one program. Since the first security class runs two programs, the turn length is doubled to accommodate the bandwidth demands of both programs. The lattice priority scheduler uses an epoch of 4 turns with a minimum of 1 turn reserved for each of H_1 and H_2 per epoch. This closely follows the configuration used for TP; assuming L has high memory demand it will consume 2 turns per epoch, and if these turns are adjacent, the dead time between them is elided, mirroring the behavior of a turn that is twice the length of the minimum.

6.6.2 Performance and Scalability

Figure 6.11 studies the performance improvement of lattice priority scheduling compared to temporal partitioning as the core count increases from 4 to 8 cores. The performance metric is system throughput normalized to temporal

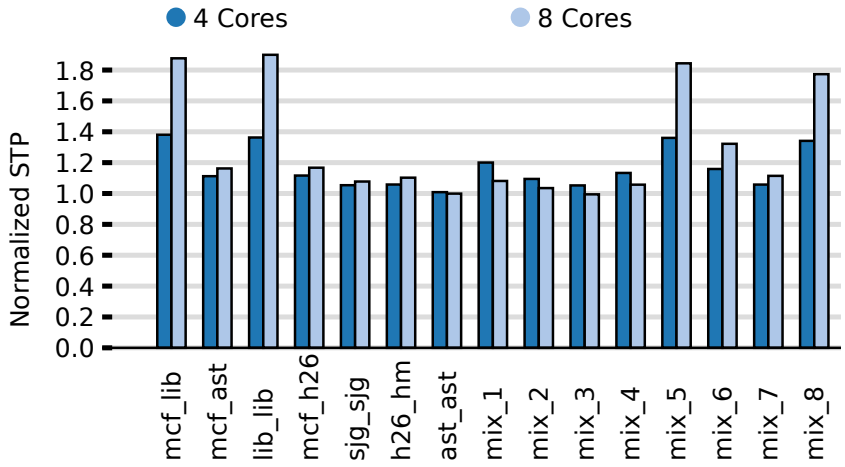

Figure 6.11: Normalized STP as core count increases.

partitioning. The 4 core machine uses the cloud computing policy described earlier, and the 8 core machine uses its natural extension. In the policy for 8 cores, Cores 1–4 share the lowest security class and Cores 5–8 are all higher than cores 1–4 and incomparable with each other. The priority scheduler uses dead time elision and decides the active class at the start of the turn. Experiments show that selecting the next active class at the start of the turn achieves better performance than selecting the next active class at the start of the dead time. In the best case (`lib_lib`), priority scheduling improves STP by 89% and 38% for 8 and 4 cores respectively. The greatest performance improvement is observed for this workload because `libquantum` is very memory intensive. On average, priority scheduling improves the STP compared to TP by 30% and 17% for the 8 and 4 core systems respectively.

6.6.3 Per-Core Performance

Figure 6.12 shows the speedup of each core individually. Each bar represents the IPC of an individual core when using LPS, normalized to the IPC of that same

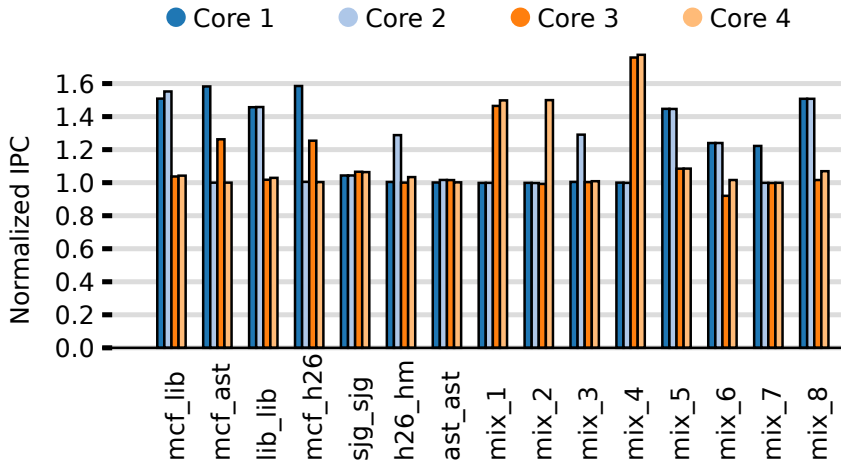

Figure 6.12: Individual core speedup.

core when using TP. Notably, the lattice-aware memory controller improves performance for cores 1 and 2 which run low-confidentiality applications as well as cores 2 and 3, which run higher-confidentiality applications. Dead time elision allows lower-confidentiality applications to continue issuing memory requests after the dead time. Priority scheduling allows the higher-confidentiality applications to use more bandwidth when the lower-confidentiality applications have few requests.

Lattice priority scheduling provides large performance improvements for some workloads in the system (77% for core 4 in `mix_4`), but only infrequently worsens the performance of other workloads. The only workload that non-negligibly reduces the IPC of one core compared to TP is `mix_6` which reduces the IPC of core 3 by 8%. However, priority scheduling still improves STP for this workload since the IPC of cores 1 and 2, are increased by over 24% each.

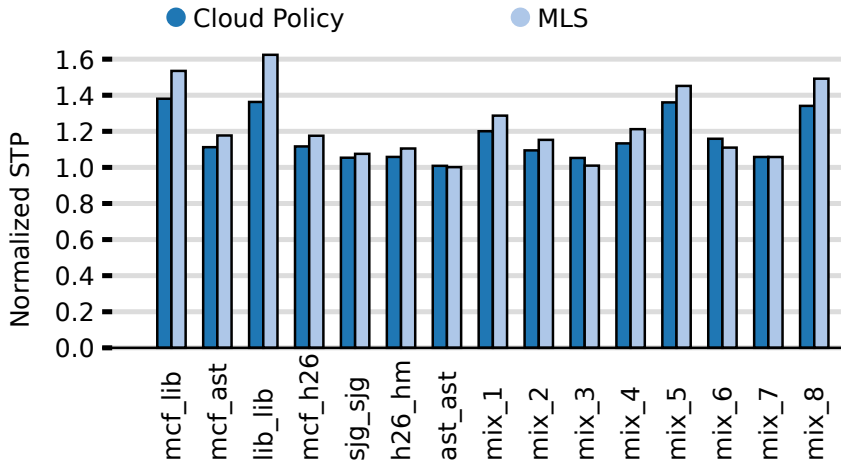

Figure 6.13: STP with two different policies normalized to the STP of the insecure baseline.

6.6.4 Lattice Policies and Performance

LPS provides more flexibility than strict static scheduling by preventing only timing channels that are specified in the policy. Therefore, performance depends on the security policy. The performance was evaluated for a 4-core system with strict TP and lattice priority scheduling using two different policies. The first policy is the cloud policy used in all other experiments. The second policy is MLS where cores 1 and 2 share the public security class. As before, TP is configured so that cores 1 and 2 share a security class.

Figure 6.13 shows the STP of lattice-aware scheduling normalized to TP. The cloud policy is more restrictive than the MLS policy since the MLS policy allows information to leak from core 3 to core 4. Therefore, with the MLS policy the improvement is higher. The MLS policy has an average improvement of 23% compared to TP and the cloud policy has an average improvement of 17%.

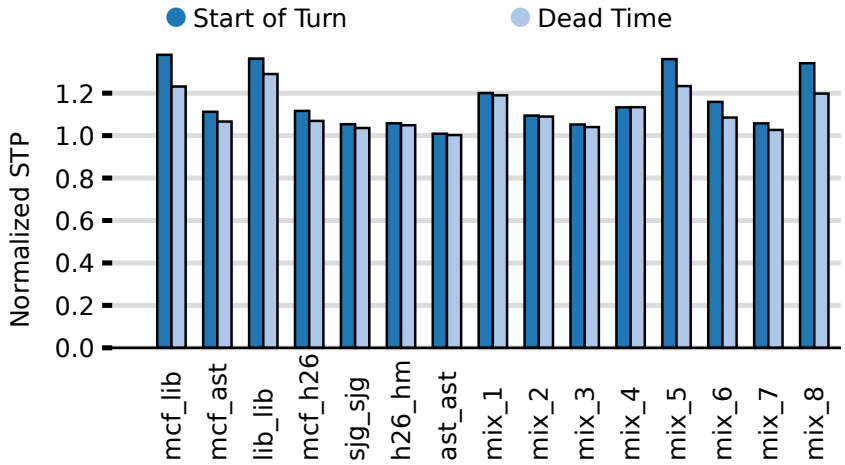

Figure 6.14: Performance impact of elision and turn allocation time.

6.6.5 Scheduling Decision Time

Figure 6.14 shows how the time when the scheduling decision is made (either at the start of the turn or at the start of the dead time) affects performance. Allocating at the dead time allows dead times to be dropped more often, but increases the chance that a lower security class will give up its turn preemptively, and have a request later in the turn get delayed. The bars represent the STP of priority scheduling when turns are allocated at the start of the turn and at the start of the dead time, each normalized to TP. For all evaluated applications, deciding which security class is scheduled next at the start of the turn is better than deciding at the start of the dead time before that turn. This is because often, the overhead of locking-in the turn allocation decision earlier is greater than the improvement gained by eliding turns more often.

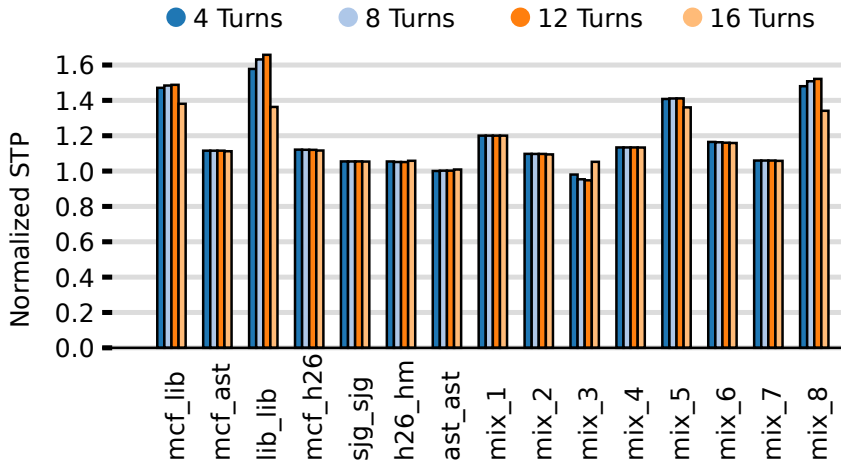

Figure 6.15: STP normalized to TP as epoch length changes.

6.6.6 Epoch Length

Figure 6.15 shows the STP of LPS as the epoch length is changed. The STP is normalized to that of the insecure baseline. The epoch length is increased from 4 to 16. Cores 3 and 4 are each given a minimum of 1 turn per epoch in all cases. This experiment shows the trade-off between providing more fairness and providing more flexibility. For some workloads, longer epochs, and therefore more flexibility, leads to better performance. Lattice priority scheduling achieves the best average STP across all workloads with an epoch length of 12. With this epoch length, the average system throughput increases by 20% and by up to 63% for `lib_lib` compared to TP. However, this increase in throughput comes at the expense of fairness. The IPC of cores 3 and 4 for `lib_lib` are reduced by 20% compared to TP. With an epoch of length 4, the scheduler is more fair, as can be seen in Figure 6.12. However, the average STP improvement is slightly lower (17%).

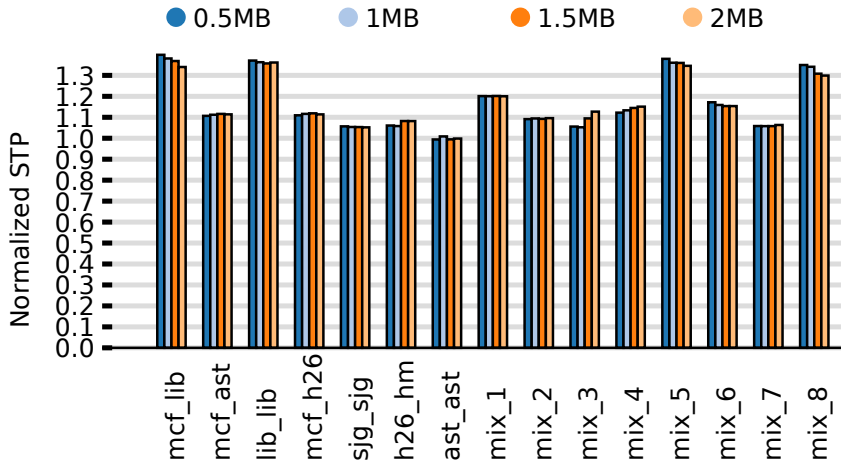

Figure 6.16: STP of lattice scheduling normalized to TP as cache per thread changes.

6.6.7 Impact of Last-Level Cache Size

The performance of both TP and lattice priority scheduling depends on the size of the LLC, since fewer cache misses means that the memory latency is incurred less often. In all experiments, private LLCs are used so that only the direct impact of our improvements to the memory controller are measured, and performance changes caused by differences in cache interference patterns are removed. Figure 6.16 shows the STP of lattice priority scheduling normalized to TP as the size of the last level cache allocated to each thread changes. Both TP and lattice priority scheduling have the same cache size. For workloads which are not very cache-sensitive, such as `lib_lib` or `ast_ast`, the size of the cache has almost no impact. In other cases, since there are fewer misses for both systems, and since the miss penalty is higher for TP, the improvement from increasing the cache size is greater for TP than for priority scheduling. The improvement of priority scheduling compared to TP is high even for larger caches.

CHAPTER 7

RELATED WORK

7.1 Gate-level information flow tracking

Gate-level information flow tracking [87, 68, 69, 85, 86, 36] applies information flow control to hardware designs at the gate-level. The earliest variations of GLIFT [87] augment each gate of the hardware implementation with additional gates to track information flow. Gate-level information flow tracking offers powerful security; all information flows including explicit flows, implicit flows, and timing channels are precisely controlled in the hardware implementation. Though it offers rather strong security, the initial GLIFT approach incurs significant area and energy overhead. To reduce these overheads, StarLogic [86] applies GLIFT logic to simulated hardware designs, obviating the need to fabricate information flow tracking logic. This reduces the overhead, but increases the development effort compared to a conventional processor design flow, because GLIFT logic expands the state-space of the hardware. Because simulating every state in large designs is intractable, prior efforts either use simulation-based GLIFT approaches to check small components [69], or limit the simulation to cover just the state-space that is reachable with software that is co-designed with the hardware [86, 85]. Using the software that executes on the hardware to drive the hardware state-space makes checking tractable while providing strong security for the software that is used during checking. However, a limitation of this approach is that it precludes updates to the firmware and operating system, and such updates are frequent and important in practice.

7.2 Hardware description languages for information flow control

More recently, security-typed hardware description languages have been developed to check that information-flow policies are enforced at design-time [50, 51, 110]. This thesis both improves upon security-typed HDLs and describes the use of such HDLs to construct secure hardware. Unlike simulation-based approaches, type systems can ensure that the entire design is secure in just seconds. To support efficient and low-area hardware designs, it is crucial for HDLs for information flow security to securely permit sharing of hardware among security domains. In the literature, there are two approaches for describing hardware that is shared: nested states, and dynamic labels.

Caisson [50] and Sapper [51] both support the description of FSMs that are shared over time using nested states. Caisson and Sapper both describe hardware as a composition of FSMs and resemble continuation-passing-style languages in which each continuation represents a state of the FSM. In these languages, nested states are continuations in which transitions from high states (in which high or low variables may be modified) to low states (in which only low variables may be modified) are permitted, yet constrained in a way that avoids violating security. The approach taken is based on the work of Zdancewic et al. [102] which studies the use of information flow control in a language with linear continuations. Linear continuations are executed exactly once along all paths in the control flow graph – for this reason, it is secure to allow them to modify high variables and return to a low context.

Nested states can be used to describe controller FSMs that multiplex between

physically separate modules for storing trusted and untrusted data. However, Caisson does not provide a mechanism for reusing registers for both trusted and untrusted data, and as a result, some registers must be duplicated in the final design. Sapper includes support for dynamic labels which permit registers to be shared over time. Sapper enforces these labels dynamically – the compiler inserts dynamic checks that convert security violations into functional correctness violations. SecVerilog [110] supports more general dependent types that are functions that are fully applied to free variables in the hardware description. The dependent types of SecVerilog can be used to describe dynamic labels. As with Caisson, the initial implementation of SecVerilog relies on dynamic enforcement of these labels. The compiler inserts hardware to clear dependently-typed registers whenever values that influence their labels are changed. This dynamic clearing mechanism is too restrictive in practice – for example, if the label of the CPU context depends on a value that can change, SecVerilog would clear all of the general purpose registers and the PC address whenever the register that influences the label changes. SecVerilog, as well as the extended type system proposed in this thesis, would benefit from supporting nested states as well, because nested state securely relax constraints on implicit flows. The type system in this thesis proposes a technique to enforce dependently-typed function labels fully statically by precisely reasoning about updates to labels on each clock edge [26]. This thesis also proposes more expressive dependent type labels than the initial version of SecVerilog [110] by adding support for function bindings which can be used to describe heterogeneously labeled arrays and bit vectors [29].

7.3 Information-flow secured processors

Much of the hardware described in this thesis has been constructed with a hardware description language with an information flow type system. Static information flow analysis of hardware provides strong assurance. There are other prior efforts on using information flow control to secure processor implementations. Tiwari et al. [87] built the first processor implementation with strong information flow guarantees at the gate level by using GLIFT logic. The processor architecture is also called GLIFT. The GLIFT architecture provides two hierarchical security domains – a trusted domain and an untrusted domain.

A particularly interesting feature of the GLIFT architecture is that it controls implicit flows as well as timing flows through branching and loop execution, while still supporting branching and loop conditions that depend on tainted values. Branching is supported securely by predicating all branches – all branch paths are executed, but only the results of the `true` path influence state changes that are committed. Loops with a fixed number of iterations are supported with a counter that is set prior to the execution of the loop.

The GLIFT architecture offers powerful security because it constrains all implicit explicit, and timing flows from the untrusted domain to the trusted domain. Meanwhile, it still offers a functional ISA capable of running benchmark programs (though it has just 19 instructions). However, GLIFT has substantial power and area overhead. All registers and memories in the implementation are duplicated, and additional taint-tracking logic is inserted. Further, the implementation does not include performance-enhancing features.

Tiwari et al. later improve upon the GLIFT architecture with the Execution

Leases [85] architecture. Instead of predication and a loop iteration counter, the Execution Leases architecture provides a new abstraction called a *lease*. Leases support branches and loops that depend on untrusted conditionals by providing a timer that bounds the execution time of the lease – the lease always returns to a trusted `PC` value when the timer expires. Leases offer better performance for branches than predication and a simpler ISA for loops than fixed iteration counters.

Tiwari et al. further improve upon the Execution Leases architecture with the Star-CPU architecture [86]. The Star-CPU architecture extends Execution Leases with a data cache that is partitioned among security domains, and a 4-stage pipeline. The Star-CPU architecture drains the pipeline and flushes CPU context before returning to the call site of the lease. Caches and pipelines are important features for reducing the average number of cycles per instruction (CPI). However, Star-CPU omits other critical performance-enhancing features including branch-prediction, branch-target prediction, TLBs, cache pipelining, a floating-point unit, and a memory controller. The processing pipeline also does not support value bypassing, and as a result, relies on the compiler to insert branch delay slots.

In addition to its CPI improvements, the Star-CPU implementation also significantly reduces the area and power overheads compared to Execution Leases and the GLIFT architecture because it is checked with StarLogic, which is simulation-based, rather than by inserting GLIFT logic in the final implementation. In addition to other optimizations, StarLogic applies GLIFT logic to simulations rather than instantiating it in hardware. Though dynamic overheads are reduced, the state-space of the simulation is increased, and checking the

state-space of an entire CPU is already intractable. To make checking with StarLogic tractable, StarCPU is co-designed with a microkernel, and only the subset of the state-space reachable with the microkernel is checked. As a result, StarCPU offers strong security for the particular microkernel that manages it, and the overheads are minimal. However, security is only offered for the exact implementation of the microkernel used for checking. Since even small changes to the software might change the reachable state-space of the hardware in unpredictable ways, kernel software and the firmware cannot be changed.

Li et al. also implement a processor with the Caisson secure HDL that they propose [51]. Because Caisson enforces security with a type system at design time, it reduces design-time overheads compared to GLIFT. Caisson also offers strong static security guarantees, namely, that well-typed hardware modules enforce timing-sensitive noninterference. Since it does not rely on state-space enumeration to enforce security, it offers security guarantees that are independent of the software, yet even complex hardware designs can be checked in a small amount of time. The Caisson processor architecture is similar to that of the Execution Leases and StarCPU architectures in that it provides a lease interface to time-multiplex the CPU among security domains. The contribution of Caisson is that it offers a way to statically check these processor techniques by providing nested states. However, the Caisson processor duplicates the entire CPU context – separate register files and program counters are kept for both the trusted and untrusted domains. Li et al. improve upon the Caisson language with the Sapper secure HDL [50]. Sapper supports fine-grained sharing of processor state such as registers by supporting dynamic labels. Dynamic labels are also enforced dynamically – the compiler automatically inserts dynamic checks that convert possible information flow violations into functional correctness er-

rors. As a result, the processor implementation with Sapper avoids duplicating the processor context.

Zhang et al. generalize the dynamic labels supported in Sapper in order to support labels that are arbitrary functions of values. The labels of SecVerilog can also be used to describe dynamic labels. SecVerilog also controls possible information flow violations caused by changes in values that influence labels through dynamic checks. Though secure, dynamic checks complicate debugging because they may cause functional errors that are not apparent by examining the code. Ferraiuolo et al. [26] extend SecVerilog to support purely static checking of labels that depend on mutable variables. They use the extended type system to construct a simple processor pipeline that includes system call handling in which system calls are induced by untrusted software yet are handled by trusted software.

The aforementioned processors all enforce a policy in which there are two hierarchical security domains. In other words, there are domains L and H, such that $L \sqsubseteq H$, but $H \not\sqsubseteq L$. These processors can therefore enforce confidentiality (H is confidential and L is trusted), or integrity (H is untrusted and L is trusted), but not both simultaneously, because the ordering on information flow for integrity and confidentiality is inverted [6]. The TrustZone-like processor described in this thesis protects both confidentiality and integrity simultaneously by providing two incomparable security domains. One domain, `CT`, is confidential and trusted, whereas the other, `PU`, is public and untrusted. The aforementioned processors also do not permit communication from H to L because doing so violates noninterference. However, such communication is needed to enforce the security policy provided by TrustZone. TrustZone permits the `CT` domain to

both read and write PU memory by downgrading loads and stores of PU memory induced when the processor is executing in the CT mode. Intuitively, the declassifications of PU memory induced by CT stores are secure because they are robust [101] – because these stores are induced by trusted software, they are not influenced by untrusted software. The TrustZone prototype implementation is also the first multi-core processor to be checked with an information flow type system, though it does not control timing channels through multi-core components.

The HyperFlow architecture further generalizes the security policies enforced by the TrustZone-like prototype by providing privilege modes and memory tags that are arbitrary information flow labels that can be represented by the lattice model of security. The HyperFlow architecture also builds upon the intuition for why the downgrades induced by CT accesses to PU memory are secure – declassifications induced by memory accesses among security domains in HyperFlow are constrained to be robust. HyperFlow also constrains endorsements through such memory accesses so they are transparent [10]. In addition to downgrades of memory in support of IPC, HyperFlow supports downgrades of registers to support passing of arguments and return values during system calls. HyperFlow permits downgrades of control-flow through a call gate mechanism. With the addition of a timer, the call gates would support the implementation of leases in software and obviate the need for downgrades. Hardware support for leases would also be a valuable extension for HyperFlow. The implementation of HyperFlow also makes contributions in terms of the performance-enhancing features it provides. These features are described in 4

Though prior information-flow secured processors control timing channels

among security domains that share a processing core over time, Timing Compartments is the first to eliminate timing channels caused by security domains that share the memory hierarchy in a multicore processor. Our empirical evaluation of Timing Compartments suggested that the memory controller was the most significant source of overhead. Lattice priority scheduling improves upon that overhead with a novel memory scheduling algorithm that more precisely enforces lattice model security policies such as those enforced by HyperFlow.

7.4 Programming Language Based Security

Programming languages for information flow control are widely studied [74]. Naturally, SecVerilog and ChiselFlow both apply techniques from software languages for information flow control. Both secure HDLs support dependent information flow labels. The use of dependent types for accurate tracking of information flow started with JFlow [64], which uses value-indexed labels. Later systems [88, 112, 49, 67, 58] have introduced more expressive forms of dependent labels, exploring trade-offs between needed expressive power and tractability of analysis. For expressive type systems to be useful in practice, type checking must be tractable and efficient for real-world code. The per-bit and per-element dependent labels this thesis describes for SecVerilog are not supported by previous dependent type systems for imperative software languages. They are an important feature for future security-typed HDLs because they offer valuable expressive power while remaining tractable: a sweet spot in the trade-off space.

The type system described in this thesis is the first language in either hardware or software to support mutable dependent information flow labels that

are enforced fully-statically at design time. Zheng et al. [113] proposed a software language that supports dynamic IFC labels in software programs. However, labels in this language are immutable. While our language is designed for hardware, we note that the proposed approach can also be applied to support mutable dependent types in imperative software languages for IFC – the reasoning that we apply to updates on registers on each clock edge can be applied to successive iterations of loops in a while-language.

Condit et al. [16] proposed Deputy, a language with a dependent type system for writing safe C programs. The dependent types in Deputy are mutable. Deputy is a general framework for dependent types with many possible applications. The authors demonstrate the use of deputy to check array bounds thereby preventing buffer overflows. The approach of Condit et al. has not been used for information flow control or for hardware. Our type system takes inspiration from this work—like Deputy, it relies on establishing the Hoare axiom of assignment wherever dependent types are updated. However, our proposed type system differs from Deputy in how it establishes this axiom and checks assignments, allowing our type system to be more expressive. In Deputy, a change to one variable requires checking all other variables with types that depend on the changed variable. We can check assignments in a more localized way — on an assignment to a variable we need only establish that the update to the assigned variable will be secure when the pending updates tracked by the next-cycle symbols are applied. The result is that in our type system, updates to variables and their types can be decoupled; in Deputy, the variable and its type must often be modified with the same parallel assignment. Our type system allows variables and their labels to be updated in different parts of the code. The ability to decouple updates to labels and the values they protect is crucial

for hardware designs – often, registers that store security information such as the privilege mode of the processor influence the security of data in desperate modules such as cache blocks.

Though many languages for information flow security intend to enforce noninterference, it is too restrictive for practical applications. As a result, all practical systems rely on downgrades to weaken noninterference. Because downgrades relax noninterference, much work has gone into strengthening assurance in languages that contain them [75]. Sabelfeld et al. [73] propose the delimited release security property which ensures that the program only releases information that the author explicitly declassifies. They study this property in the context of an imperative software language, and show that it prevents laundering attacks which would otherwise enable attackers to cause more information to be released than intended. To typecheck the TrustZone-like prototype in this paper, we extended SecVerilog with a system for downgrading that is similar to the one proposed by Sabelfeld and Myers [75].

The assurance offered by a system that enforces the delimited release property depends on the implementation details of the system because released information is limited to the downgrades in the implementation. By contrast, *extensional* security properties are purely defined in terms of the behavior of the system, and offer value without inspecting the implementation of the system. Robust declassification [101] is an extensional property that prevents low-integrity attackers from influencing what information is declassified. It is specified as a 4-safety hyperproperty that holds of a system if for one attack (low-integrity input) the public outputs are indistinguishable for any two secret inputs, then there is no other attack that will cause public outputs to differ based

on the choice of secret inputs. In other words, the choice of low-integrity inputs has no affect on what information released, so a low-integrity attacker could not have influenced declassification. Dually, transparent endorsement [10] is an extensional 4-safety hyperproperty that prevents parties from endorsing information that they could not have read. Non-malleable information flow control is a hyperproperty that is a superset of robust declassification and transparent endorsement. ChiselFlow and HyperFlow both restrict information flows in a similar way as systems that were constructed to study the non-malleable information flow control security property.

HyperFlow intends to support the policies expressed in languages for information flow control. The decentralized label model (DLM) [65, 66] permits the expression of information flow policies involving principals that are mutually distrusting, but communicate. The DLM is valuable for systems that are decentralized such as distributed systems and microkernels. In the DLM, data is associated with principals that have authority over (or equivalently privilege to read or write) data. The ability to release information is restricted to principals that have authority over the data. Chong et al. formalize this idea by generalizing robust declassification to the DLM [15]. Other label models that support decentralized information flow control in the context of microkernel operating systems [24, 104] and distributed systems [105, 56, 14] have also been proposed.

The DLM relies on an underlying authorization mechanism to permit information release. Authorization within DLM borrows conceptually from earlier work on authorization logics [77, 44] in which a `speaks for` relation is used to permit principals to delegate authority. In the context of the Fabric distributed system for information flow security, Liu et al. [57] define a trust ordering that

is perpendicular to the information flow ordering. Interestingly, unlike in the information flow ordering, integrity and confidentiality are not inverted in the trust ordering [6]. The flow-limited authorization logic (FLAM) [2] builds on this notion of a trust ordering and unifies authorization logics and languages for information flow control. In FLAM, labels and information flow labels are the same syntactic objects. Principals are ordered by an information flow order and an acts-for order which is conceptually similar to the `speaks for` relation in authorization logics. We chose the FLAM model as the first label model to represent with the hypercube labels of HyperFlow. Because FLAM represents authority and information flow policies with the same syntactic objects, translations of FLAM labels into hypercube labels are compact in terms of the number of required bits.

7.5 Information-flow tracking architectures

HyperFlow enforces security by associating security tags representing information flow labels with memory pages and registers, and by tracking the propagation of these tags. DIFT is the earliest architecture for information flow tracking [82]. DIFT associates 1-bit tags with memory addresses that indicates whether or not they are trusted. DIFT then dynamically propagates these tags and prevents insecure information flows. The Raksha [20] architecture improves on DIFT by allowing programmer-defined tag-propagation policies and providing tag violation exceptions. PUMP [22] enforces flexible tag propagation policies. Like HyperFlow, the Loki [106] processor enforces application-defined information flow policies in hardware. However, none of the implementations of these processors have been secured with an HDL that is augmented with

an information flow type system. In Loki, information flow labels are opaque pointers to objects that represent those labels. Loki caches the translations from pointers to labels. HyperFlow represents information flow policies in a way that permits inspection of the labels at the hardware-level. This allows access controls that enforce these policies to be checked in a way that is amenable to verification with the information flow type system and is efficient because it does not require software intervention to perform label computations.

7.6 Capability-Based Addressing

HyperFlow provides memory protection by using information flow labels. Hardware architectures for capability-based addressing also provide an alternative to conventional memory protection [9, 100, 95]. Capabilities are an approach for controlling authorization in computer systems that have been broadly applied to many domains in computing [48]. Access control lists are an authorization mechanism in which a list of principals authorized to perform some operation on an object is associated with that object. By contrast, capabilities permit authorization by associating a list of operations that can performed on objects with principals. Capabilities must be unforgeable, but can typically be transferred among principals.

Capabilities-based addressing is a hardware technique for controlling the authority of a process to read or write a memory location [9]. In a machine that provides capabilities-based addressing, loads and stores are guarded by capabilities. These capabilities are pointers that specify a base address, a length, and permissions such as read, write, and execute. An advantage of capabil-

ities compared to more conventional virtual memory based protection is that capabilities provide more fine-grained protection. Capabilities can often offer distinct permissions for a process for each unique word of memory whereas virtual memory offers distinct permissions for each page. By providing word-granularity protection, capabilities-based addressing can defend against vulnerabilities such as buffer overflows in unsafe programming languages like C [9, 100].

Woodruff et al. [100, 95] propose the CHERI architecture and processor implementation for capabilities-based addressing. CHERI supports both capabilities as well as conventional memory protection with a hardware MMU. As a result, CHERI supports incremental adoption of capabilities. For example, CHERI supports systems in which libraries provide protection with capabilities, and can be utilized by capability-unaware legacy applications thereby improving security for unmodified applications. Supporting legacy applications is also a goal of HyperFlow, and HyperFlow similarly supports the interoperability of memory protection through information flow control and virtualization. Unlike CHERI, the threat-model of HyperFlow also considers timing channel attacks and defends against timing channels in both the TLBs and hardware page-table walkers.

HyperFlow and CHERI are micro-architecturally similar in that both provide memory protection through tagged physical addresses. However, tags in HyperFlow represent information flow labels rather than capabilities, and as a result, provide stronger security. Capabilities protect data only at the site where a principal is authorized to access that data. By contrast, information flow control constrains the movement of data even after principals have been autho-

rized to receive that data. Information flow control also defends attacks that systems protected by capabilities cannot prevent. Though it has been widely believed that capabilities can defend against confused deputy attacks, it has recently been shown that capabilities cannot defend against all such attacks [71]. By contrast, integrity protection though information flow control can defend against confused deputy attacks.

The prototype implementation of HyperFlow provides protection at the granularity of memory pages rather than providing finer-grained protection by associating tags with each word of memory. As a result, the prototype is less equipped to defend against attacks on memory-safe languages. As in any hardware mechanism for memory protection, there is a trade-off between granularity of protection and memory density because protection information also requires storage. Witchel et al. study this trade-off [96]. Watson et al. [95] note that another advantage of fine-grained protection is the ability to provide heterogeneous security policies for data structures and objects. During the development of an operating system for CHERI, Watson et al. augmented CHERI with hardware support for object capabilities. Preliminary work on developing an operating system for HyperFlow suggests that heterogeneous protection for objects and data structures in the operating system would benefit HyperFlow as well. We chose page-granularity tags because doing so reduces the memory overhead. However, changing the granularity of protection does not fundamentally change the architecture or impose challenges with regard to static label-checking. Prior architectures for dynamic information flow tracking propose multi-granularity tagging which permits a mix of page-granularity and word-granularity tags to coexist at runtime [82, 106]. Multi-granularity tags are naturally an optimization that should be considered for implementations of Hy-

perFlow. However, label-checking multi-granularity tags would be non-trivial because doing so would require policies that change at runtime.

7.7 Enclave Architectures

Processor architectures such as Intel SGX [17, 18, 8] as well as academic efforts [19, 11, 1] provide protection for software modules called *enclaves*. Although a mostly unmodified operating system still manages resources such as virtual memory and CPU time for the enclaves, the confidentiality and integrity of the enclaves is still protected even if the OS is malicious or compromised. The high-level approach taken by these architectures is to use a *reference monitor* that allows the operating system to construct and manage enclaves, but rejects operations that might violate the confidentiality or integrity of the enclaves. The reference monitor is typically implemented with instruction extensions for managing enclaves. Hyperwall [83] protects enclaves in a system managed by a hypervisor. Ascend [30] prevents physical attacks including side channels that are exploitable by adversaries that can measure memory access patterns. Ascend prevents these attacks by relying on oblivious RAMs [34, 81].

Software implementations of enclaves have also been proposed [61, 25]. At minimum, software implementations of enclaves require hardware support that includes storage that is inaccessible to the operating system to contain enclave data and resource management data, and optionally, support for attestation and defending against physical attacks if needed [25]. Costan et al. note that Intel SGX is implemented as primarily with microcode, and in a sense, is thus already software [18].

Most prior architectures for enclaves do not defend against side-channel attacks. Sanctum prevents cache timing channels [19], but other timing channels such as those caused by branch prediction are not addressed. Timing channel attacks that exploit branch predictors have been used to attack Intel SGX enclaves [46]. HyperFlow enforces information flow policies in a timing-safe way, intending to eliminate microarchitectural timing channel attacks. Though timing channel attacks are subtle, by implementing HyperFlow in an HDL for information flow control, we provide strong assurance that we have succeeded in eliminating them.

Enclave management software implemented for HyperFlow would provide stronger assurance than prior enclave management systems by eliminating timing channels. HyperFlow augments virtual memory protection with information flow control memory protection that operates on physical pages of memory. As a result, an enclave management system for HyperFlow can be implemented by storing enclave data and enclave resource allocation data in memory protected by labels that are inaccessible to the OS. Komodo implements enclaves on ARM TrustZone by executing enclaves and enclave management software in TrustZone's secure world and by executing the untrusted OS and all other software in the normal world [25]. Roughly, secure world memory is confidential and trusted and normal world memory is public and untrusted [29]. HyperFlow can emulate TrustZone because it provides memory protection with generalized information flow labels that can describe confidentiality and integrity policies. By implementing enclave management software for HyperFlow, stronger security is provided even for systems managed by unmodified legacy operating systems and applications.

Aside from microarchitectural timing channels, there are other side channels that enclave systems do not address, and are perhaps better addressed by the operating system. For example, enclave systems do not address *passive address translation attacks* in which the operating system observes the enclave's page faults and correlates the enclave's memory usage with secrets [18]. Active attacks in which the operating system exploits side channels have also been demonstrated, including Iago attacks [12] in which the operating system returns spurious values in response to system calls, and controlled channel attacks in which the operating system deliberately induces page faults. Because enclave systems do not prevent side channels, the operating system is ultimately still trusted to protect the enclave's confidentiality.

7.8 Timing Channels

7.8.1 Timing Channel Attacks

Many microarchitectural timing channels have been identified. Vulnerabilities have also been found in caches [55, 70, 5], branch predictors [47, 46], processor pipelines [92], networks on chip [91, 94], and memory controllers [90, 31]. Recently, the Spectre [42] and Meltdown [52] vulnerabilities have been demonstrated to exploit speculative and out-of-order execution in Intel processors. Meltdown is especially devastating because it can be used to leak arbitrary kernel data. During our study of Timing Compartments, we identified new timing channels in cache coherence protocols.

7.8.2 Timing channel defense in hardware

Temporal and spatial partitioning techniques are most amenable to information flow control analysis because they provide isolation, and therefore likely enforce noninterference. For example, cache partitioning can be used to prevent timing channels caused by contention among distrusting processes. Liu et al. [53] demonstrated that way partitioning intended for performance isolation in conventional Intel processors can also be used for timing channel protection.

Timing compartments leverage prior proposals which use temporal partitioning to protect the network [94, 91] and memory controller [90]. Shaifee et al. [78] propose rank partitioning and the triple alternation to improve performance of temporally partitioned memory controllers. Both optimizations can be added to timing compartments. However, the previous study [78] showed that triple alternation has comparable performance with bank partitioning used in timing compartments. Rank partitioning requires significant restrictions to memory allocation.

This thesis describes an approach for efficiently preventing memory controller timing channels by precisely enforcing security policies expressed in the lattice model [28]. Hu [37] proposed lattice scheduling, which uses the lattice model to efficiently address timing channels in process schedulers. Lattice scheduling schedules processes so they increase monotonically in security level in order to avoid flushing caches when context switching from one process to another with a higher class. Lattice scheduling was an inspiration for the dead time elision technique used in lattice priority scheduling.

Other approaches for cache timing-channel protection have also been pro-

posed. For example, RPCache [93] and Random Fill cache [54] obfuscate cache timing by randomizing cache replacements and insertions respectively. These do not enforce noninterference, however, and are thus not amenable to information flow analysis. Kong et al. also demonstrate that RPCache is vulnerable to attacks [43]. NoMoCache [23] partitions some cache ways, but allows interference in other cache ways to reduce timing channel capacity. This approach attempts to trade-off security in favor of better performance, though it is unclear what security is ultimately offered in general.

Preventing timing channels through information flow analysis offers formal guarantees, namely that a timing-safe variant of noninterference is enforced. Another promising approach to the design of hardware with clear security guarantees is by applying the field of quantitative information flow control [80]. Quantitative information flow control uses information theory to bound the amount of information leaked through a side-channel such as a timing channel. Fletcher et al. apply quantitative information flow control to bound the information leaked in an intra-program timing channel in memory controllers [31].

In addition techniques for preventing and mitigating timing channels, others have proposed techniques for detecting timing channel. Hunger et al. [38] formally model timing channels, and demonstrate that *reads* from a covert timing channel are destructive – they mutate state which causes interference. This interference facilitates detection since it implies reads can be observed. They also show how attacks can both be performed and detected even through noisy channels. Chen et al. [13] propose CC-Hunter, a framework for timing channel detection that uses hardware support to detect bursts of events that are likely to correspond to attempts to use a timing channel.

CHAPTER 8

CONCLUSON

8.1 Summary

HDLs for information flow control are a promising way to ensure that hardware implementations are secure. This thesis has made contributions towards the development of HDLs for information flow security and the development of hardware that can be proven secure with the use of information flow control HDLs.

It improves upon secure HDLs by adding support for heterogeneously labeled data structures, by fully-statically enforcing dynamic information flow labels, by supporting downgrades that relax information flow control, and by supporting label inference. The data structures that can be labeled heterogeneously include arrays, bit vectors, and structs (or records). Arrays and bit vectors are supported through dependent labels that include function bindings. For bit vectors, the function binding represents a bit index into the vector. For arrays, the labels are curried functions with two bindings. The first binding is an index into the array, and applying that function to the indexing expression produces a bit vector label with one binding that represents an index into the vector as before.

Mutable dependent labels are supported fully statically by a type system that constrains updates to dependently-labeled variables based on an updated version of the label. In an HDL, assignments to sequential variables represent an update that will take place at the start of the next clock edge. Therefore, that

assignment should be checked using the new valuation of that variable's label on that cycle. The new valuation of the label is computed by replacing occurrence of variables in the label with the combinational inputs to those variables. In doing so, the type system can ensure at design time that updates to the labels of variables are secure.

Downgrades are necessary in all practical systems for information flow control. Because downgrades relax noninterference, effort has been made to ensure that downgrades do not cause harm. This thesis applies prior approaches to securing downgrades called robust declassification [101] and transparent endorsement [10]. Robust declassification prevents untrusted code from influencing whether or not information is declassified. Transparent endorsement prevents a party from endorsing data that it could not have read. Both forms of downgrades impose constraints on the confidentiality and integrity of the data being downgraded and the context in which it is downgraded. Therefore the language must have notions of confidentiality and integrity. The HDL proposed in this thesis uses product labels that are pairs of confidentiality and integrity labels. Robust and transparent downgrades are then enforced in a similar way to software languages.

This thesis studies the use of a security-typed HDL to secure a processor architecture that resembles ARM TrustZone. This study shows that the overhead of security-typed HDLs is minimal in terms of area, power, frequency, and CPI. We also recreated security vulnerabilities found in commercial processors and have shown that the security-typed HDL is able to detect them.

This thesis proposes a novel architecture for information flow security, called HyperFlow. HyperFlow replaces conventional privilege levels and memory

protection with information flow labels. In doing so, it offers better separation of privilege, for example, in a system managed by a microkernel. HyperFlow also supports communication across security domains during IPC and system calls. HyperFlow is implemented in a security-typed HDL providing strong assurance about its implementation. HyperFlow also prevents timing channels through micro-architectural components that are shared over time in a single processor core.

We also study a multi-core architecture for timing channel security called Timing Compartments. Timing Compartments relies on simple temporal and spatial partitioning mechanisms to prevent timing channels so that it is amenable to information flow verification by a secure HDL. Timing compartments also proposes performance optimizations including coordinated scheduling of time-multiplexed resources, command-aware memory controller dead time, and application-aware allocation of resources.

Finally, this thesis has proposed lattice priority scheduling (LPS) to improve the performance of timing-safe memory controllers. Our performance results about the timing compartments architecture suggest that memory controller timing channel protection is the performance bottleneck. LPS improves upon a straight-forward time-multiplexed memory controller by more precisely enforcing security policies in the lattice model of security. In doing so, LPS both improves upon the total available memory bandwidth and better responds to the run-time behavior of applications without weakening security.

8.2 Future Directions

8.2.1 Secure HDLs

A benefit of information flow security for HDLs is that they enforce timing-sensitive noninterference because HDLs give a cycle-level description of the hardware. Timing channels in hardware implementations become implicit flows that are explicit in the HDL code that implements the hardware. As a result, timing flows in hardware implementations are also difficult to differentiate from other kinds of flows. Differentiating timing flows from other information flows in hardware designs is valuable. Many hardware designs do not consider timing channels to be threats because timing channel attacks have historically been more difficult to exploit than other kinds of attacks, and timing channel protection often comes with substantial performance penalties. It may also be desirable to address some timing channels in software rather than hardware. It may be easier to analyze what information is released by a timing flow in software, and therefore avoid constraining timing flows in hardware.

8.2.2 Secure Hardware

The enforcement mechanisms in HyperFlow resemble dynamic information flow architectures. Preventing implicit flows with dynamic information flow control has historically been difficult. However, it may be plausible that HyperFlow can control them because it propagates a notion of the program counter label to the hardware and it may be possible to constrain updates to the program counter label. HyperFlow permits updates to the program counter label

as long as they are within a range. Movement within this range is necessary to permit robust and transparent downgrades of registers and memory locations. However, it may be possible to avoid using a range by relaxing the constraints on downgrades so that just the confidentiality or integrity aspects of the program counter label are used. Changes to the program counter label would then only be possible through control gates, which can also be constrained so that they are robust and transparent.

HyperFlow represents memory protection and privilege levels as information flow labels that are physically implemented as small, finite bit-vectors. However, practical systems involve many mutually distrusting processes. As a result, virtualization of these labels will be a crucial feature for a practical implementation

Because HyperFlow enforces information flow security and constrains downgrades so that they are robust and transparent, it is naturally desirable to have a proof of a security property about the HyperFlow ISA. Naturally, this security property is likely to resemble the non-malleable information flow control property. An interesting aspect of such a security proof is that the ISA supports control flow through jump and branch instructions, which may be difficult to reason about mathematically.

8.2.3 An Operating System for HyperFlow

A natural future direction for HyperFlow is the design of an operating system that manages HyperFlow. The software that manages security labels in HyperFlow must be trusted. However, because the HyperFlow ISA offers support

for fine-grained information flow policies, a natural design for the operating system is a microkernel in which components of the operating system are protected with information flow labels. Such an operating system would offer better separation of privilege than a conventional microkernel, because ideally, the only component of the operating system for HyperFlow that would need to be universally trusted is the subset that manages information flow labels.

An interesting comparison point to a future operating system for HyperFlow is the Intel SGX architecture [17, 18], because in some sense, SGX resembles a microkernel that is implemented in firmware and microcode. Microkernels offer better separation of privilege compared to a monolithic kernel. Micorkernels are compartmentalized and many components of a microkernel, such as device drivers, execute in user-space. Compromise of kernel components that execute in userspace are less harmful because these components have a restricted view of memory, unlike supervisor-mode software that can access all of physical memory. However, conventional microkernels must execute components that are responsible for resource management, such as process scheduling and virtual memory management in supervisor mode because conventional processors support only a small number of purely hierarchical privilege levels. Systems protected by Intel SGX, however, do not place virtual memory management or process scheduling in the trusted computing base. SGX prevents compromise of the integrity and confidentiality of applications through a small reference monitor that controls a region of memory that is inaccessible even to software with supervisor privilege. The operating system is still permitted to manage the virtual memory of enclaves by communicating with that reference monitor via enclave management instructions. Removing resource management components of the operating system from the trusted computing base is a considerable

improvement because resource management is likely to be vulnerable subset of the operating system code. Resource-management is performance critical code. Performance critical code is difficult to implement correctly because it is often complex.

An operating system that manages HyperFlow can potentially offer both greater security and performance than either a system protected by SGX or a conventional microkernel. Because HyperFlow offers lattice-model privilege levels, it can offer more fine-grained protection for microkernel components. It can also support system calls which are taken directly from an application to a label delegated to a small component of the operating system such as a network driver. In doing so, there are fewer transitions in privilege, and performance can be improved.

BIBLIOGRAPHY

[1]

[2] O. Arden and A. C. Myers. A Calculus for Flow-Limited Authorization. In *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*, June 2016.

[3] Aslan Askarov, Danfeng Zhang, and Andrew C. Myers. `Predictive Black-Box Mitigation of Timing Channels`. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, 2010.

[4] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. AviÅ¿ienis, J. Wawrzynek, and K. AsanoviÄĞ. Chisel: Constructing hardware in a scala embedded language. In *DAC Design Automation Conference 2012*, 2012.

[5] Daniel J. Bernstein. `Cache-Timing Attacks on AES`. Technical report, 2005.

[6] K. J. Biba. Integrity Considerations for Secure Computer Systems. Technical Report ESD-TR-76-372, USAF Electronic Systems Division, 1977.

[7] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. `The Gem5 Simulator`. *SIGARCH Computer Architecture News*, 2011.

[8] Rick Boivie. SecureBlue++: CPU Support for Secure Execution, 2012.

[9] Nicholas P. Carter, Stephen W. Keckler, and William J. Dally. Hardware Support for Fast Capability-based Addressing. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VI, 1994.

[10] Ethan Cecchetti, Andrew C. Myers, and Owen Arden. Nonmalleable Information Flow Control. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17.

[11] David Champagne. *Scalable Security Architecture for Trusted Software*. PhD thesis, Princeton University, 2010.

[12] Stephen Checkoway and Hovav Shacham. Iago attacks: Why the system call api is a bad untrusted rpc interface. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, 2013.

[13] Jie Chen and Guru Venkataramani. `CC-Hunter: Uncovering Covert Timing Channels on Shared Processor Hardware`. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014.

[14] Winnie Cheng, Dan R. K. Ports, David Schultz, Victoria Popic, Aaron Blankstein, James Cowling, Dorothy Curtis, Liuba Shrira, and Barbara Liskov. Abstractions for usable information flow control in aeolus. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, 2012.

[15] Stephen Chong and Andrew C. Myers. Decentralized robustness. In *19th IEEE Computer Security Foundations Workshop (CSFW)*, 2006.

[16] Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George C. Necula. Dependent Types for Low-level Programming. ESOP'07.

[17] Intel Corporation. Intel Software Guard Extensions Programming Reference, 2014.

[18] Victor Costan and Srinivas Devadas. Intel SGX explained. Technical Report http://eprint.iacr.org/2016/086, February 2016.

[19] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *25th USENIX Security Symposium (USENIX Security 16)*, 2016.

[20] Michael Dalton, Hari Kannan, and Christos Kozyrakis. Raksha: A Flexible Information Flow Architecture for Software Security. ISCA '07, 2007.

[21] Dorothy E. Denning. A Lattice Model of Secure Information Flow. *Communications of the ACM*, 1976.

[22] Udit Dhawan, Nikos Vasilakis, Raphael Rubin, Silviu Chiricescu, Jonathan M. Smith, Thomas F. Knight, Jr., Benjamin C. Pierce, and André DeHon. Pump: A programmable unit for metadata processing. HASP '14, 2014.

[23] Leonid Domnitser, Aamer Jaleel, Jason Loew, Nael Abu-Ghazaleh, and Dmitry Ponomarev. `Non-Monopolizable Caches: Low-Complexity Mitigation of Cache Side Channel Attacks`. *ACM Transactions Architecture and Code Optimization*, 2012.

[24] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and event processes in the asbestos operating system. SOSP '05, 2005.

[25] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, 2017.

[26] Andrew Ferraiuolo, Weizhe Hua, Andrew C Myers, and G Edward Suh. Secure information flow verification with mutable dependent types. In *Proceedings of the 54th Annual Design Automation Conference 2017*, page 6. ACM, 2017.

[27] Andrew Ferraiuolo, Yao Wang, Rui Xu, Andrew C. Myers, and G. Edward Suh. Full-Processor Timing Channel Protection with Applications to Secure Hardware Compartments. Technical Report http://hdl.handle.net/1813/41218, Cornell University, 2015.

[28] Andrew Ferraiuolo, Yao Wang, Danfeng Zhang, Andrew C. Myers, and G. Edward Suh. `Lattice priority scheduling: Low-overhead timing-channel protection for a shared memory controller`. In *IEEE International Symposium on High Performance Computer Architecture*, 2016.

[29] Andrew Ferraiuolo, Rui Xu, Danfeng Zhang, Andrew C. Myers, and G. Edward Suh. Verification of a practical hardware security architecture through static information flow analysis. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17.

[30] Christopher W. Fletcher, Marten van Dijk, and Srinivas Devadas. A Secure Processor Architecture for Encrypted Computation on Untrusted Programs. In *Proceedings of the Workshop on Scalable Trusted Computing*, 2012.

[31] Christopher W. Fletcher, Ling Ren, Xiangyao Yu, Marten van Dijk,

Omer Khan, and Srinivas Devadas. Suppressing the Oblivious RAM Timing Channel While Making Information Leakage and Program Efficiency Trade-Offs. In *20th IEEE International Symposium on High Performance Computer Architecture*, 2014.

[32] Vijay K. Garg. *Introduction to Lattice Theory with Computer Science Applications*. Wiley Publishing, 2015.

[33] J.A. Goguen and J. Meseguer. Security Policies and Security Models. In *IEEE S&P*, 1982.

[34] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 1996.

[35] Matthew Hicks, Cynthia Sturton, Samuel T. King, and Jonathan M. Smith. SPECS: A Lightweight Runtime Mechanism for Protecting Software from Security-Critical Processor Bugs. In *ASPLOS*, 2015.

[36] Wei Hu, Dejun Mu, Jason Oberg, Baolei Mao, Mohit Tiwari, Timothy Sherwood, and Ryan Kastner. Gate-level information flow tracking for security lattices. *DAES*, 2014.

[37] Wei-Ming Hu. Lattice Scheduling and Covert Channels. In *Proceedings of the Symposium on Security and Privacy*, 1992.

[38] Casen Hunger, Mikhail Kazdagli, Ankit Singh Rawat, Alexandros G. Dimakis, Sriram Vishwanath, and Mohit Tiwari. Understanding Contention-Based Channels and Using Them for Defense. In *Proceedings of the International Symposium on High Performance Computer Architecture*, 2015.

[39] Intel Corporation. http://ark.intel.com/compare/84679,84678,84677,84676.

[40] Colin J. Fidge. Timestamps in message-passing systems that preserve partial ordering. 1988.

[41] Adam N. Jacobvitz, Andrew D. Hilton, and Daniel J. Sorin. Multi-Program Benchmark definition. In *International Symposium on Performance Analysis of Systems and Software*, 2015.

[42] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and

Yuval Yarom. Spectre attacks: Exploiting speculative execution. *ArXiv e-prints*, January 2018.

[43] Jingfei Kong, Onur Aciicmez, Jean-Pierre Seifert, and Huiyang Zhou. Deconstructing New Cache Designs for Thwarting Software Cache-based Side Channel Attacks. In *Proceedings of the 2Nd ACM Workshop on Computer Security Architectures*, CSAW '08, 2008.

[44] Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: Theory and practice. *ACM Trans. Comput. Syst.*, 10(4):265–310, November 1992.

[45] L. J. LaPadula and D. E. Bell. Secure Computer Systems: A Mathematical Model. 1996.

[46] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 557–574, Vancouver, BC, 2017. USENIX Association.

[47] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 557–574, Vancouver, BC, 2017. USENIX Association.

[48] Henry M. Levy. *Capability-Based Computer Systems*. Butterworth-Heinemann, Newton, MA, USA, 1984.

[49] Peng Li and Steve Zdancewic. Downgrading policies and relaxed noninterference. Long Beach, CA, January 2005.

[50] Xun Li, Vineeth Kashyap, Jason K. Oberg, Mohit Tiwari, Vasanth Ram Rajarathinam, Ryan Kastner, Timothy Sherwood, Ben Hardekopf, and Frederic T. Chong. Sapper: A Language for Hardware-level Security Policy Enforcement. In *ASPLOS*, 2014.

[51] Xun Li, Mohit Tiwari, Jason K. Oberg, Vineeth Kashyap, Frederic T. Chong, Timothy Sherwood, and Ben Hardekopf. Caisson: A Hardware Description Language for Secure Information Flow. In *PLDI*, 2011.

[52] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner

Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *ArXiv e-prints*, January 2018.

[53] Fangfei Liu, Qian Ge, Yuval Yarom, Carlos Mckeen, Frank Rozas, Gernot Heiser, and Ruby Lee. `CATalyst: Defeating Last-Level Cache Side Channel Attacks in Cloud Computing`. In *Proceedings of the 22nd International Symposium on High Performance Computer Architecture*, 2016.

[54] Fangfei Liu and Ruby B. Lee. `Random Fill Cache Architecture`. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014.

[55] Fangfei Liu, Y. Yarom, Qian Ge, G. Heiser, and R.B. Lee. `Last-Level Cache Side-Channel Attacks are Practical`. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2015.

[56] Jed Liu, Owen Arden, Michael D. George, and Andrew C. Myers. Fabric: building open distributed systems securely by construction. 2017.

[57] Jed Liu, Michael D. George, K. Vikram, Xin Qi, Lucas Waye, and Andrew C. Myers. Fabric: A platform for secure distributed computation and storage. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, 2009.

[58] Luísa Lourenço and Luís Caires. Dependent information flow types. In *POPL*, 2015.

[59] ARM Ltd. ARM Security Technology: Building a Secure System using TrustZone Technology.

[60] Robert Martin, John Demme, and Simha Sethumadhavan. Timewarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. *SIGARCH Comput. Archit. News*, 40(3):118–129, June 2012.

[61] Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: An execution infrastructure for tcb minimization. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, Eurosys '08, 2008.

[62] CVE-2017-5691, July 2017.

[63] A. Myers. Mostly-static decentralized information flow control. Technical report, Cambridge, MA, USA, 1999.

[64] Andrew C. Myers. JFlow: Practical Mostly-static Information Flow Control. In *POPL*.

[65] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *16th ACM Symp. on Operating System Principles (SOSP)*, October 1997.

[66] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Trans. Softw. Eng. Methodol.*, 2000.

[67] Aleksandar Nanevski, Anindya Banerjee, and Deepak Garg. Verification of information flow and access control policies with dependent types. pages 165–179, 2011.

[68] Jason Oberg, Wei Hu, Ali Irturk, Mohit Tiwari, Timothy Sherwood, and Ryan Kastner. Theoretical Analysis of Gate Level Information Flow Tracking. In *DAC*, 2010.

[69] Jason Oberg, Wei Hu, Ali Irturk, Mohit Tiwari, Timothy Sherwood, and Ryan Kastner. Information Flow Isolation in I2C and USB. In *DAC*, 2011.

[70] Colin Percival. `Cache Missing for Fun and Profit`. In *BSDCan*, 2005.

[71] V. Rajani, D. Garg, and T. Rezk. On Access Control, Capabilities, Their Equivalence, and Confused Deputy Attacks. In *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*, 2016.

[72] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. `DRAMSim2: A Cycle Accurate Memory System Simulator`. *Computer Architecture Letters*, 2011.

[73] Andrei Sabelfeld and Andrew C. Myers. A Model for Delimited Information Release. In *IEEE S&P*.

[74] Andrei Sabelfeld and Andrew C. Myers. Language-based Information-flow Security. *IEEE Journal on Selected Areas in Communications*, 2006.

[75] Andrei Sabelfeld and David Sands. Declassification: Dimensions and Principles. *JCS*, 2009.

[76] Jerome H. Saltzer. Protection and the control of information sharing in multics. *Commun. ACM*, 1974.

[77] Fred B. Schneider, Kevin Walsh, and Emin Gün Sirer. Nexus authorization logic (nal): Design rationale and applications. *ACM Trans. Inf. Syst. Secur.*, 2011.

[78] Ali Shafiee, Akhila Gundu, Manjunath Shevgoor, Rajeev Balasubramonian, and Mohit Tiwari. Avoiding Information Leakage in the Memory Controller with Fixed Service Policies. In *Proceedings of the International Symposium on Microarchitecture*, 2015.

[79] Sergei Skorobogatov and Christopher Woods. Breakthrough Silicon Scanning Discovers Backdoor in Military Chip. In *CHES*, September 2012.

[80] Geoffrey Smith. On the foundations of quantitative information flow. In *Proceedings of the 12th International Conference on Foundations of Software Science and Computational Structures: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, FOSSACS '09, 2009.

[81] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path oram: An extremely simple oblivious ram protocol. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, 2013.

[82] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XI, 2004.

[83] Jakub Szefer and Ruby B. Lee. `Architectural Support for Hypervisor-Secure Virtualization`. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages*, 2012.

[84] David Lie Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. `Architectural Support for Copy and Tamper Resistant Software`. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000.

[85] Mohit Tiwari, Xun Li, Hassan M. G. Wassel, Frederic T. Chong, and Timo-

thy Sherwood. Execution Leases: A Hardware-Supported Mechanism for Enforcing Strong Non-Interference. In *MICRO*, 2009.

[86] Mohit Tiwari, Jason K. Oberg, Xun Li, Jonathan Valamehr, Timothy Levin, Ben Hardekopf, Ryan Kastner, Frederic T. Chong, and Timothy Sherwood. Crafting a Usable Microkernel, Processor, and I/O System with Strict and Provable Information Flow Security. In *ISCA*, 2011.

[87] Mohit Tiwari, Hassan M.G. Wassel, Bita Mazloom, Shashidhar Mysore, Frederic T. Chong, and Timothy Sherwood. Complete Information Flow Tracking from the Gates Up. In *ASPLOS*, 2009.

[88] Stephen Tse and Steve Zdancewic. Run-time principals in information-flow type systems. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2004.

[89] Yao Wang, Andrew Ferraiuolo, and Edward Suh. `Timing Channel Protection for a Shared Memory Controller`. In *Proceedings of the 20th International Symposium on High Performance Computer Architecture*, 2014.

[90] Yao Wang, Andrew Ferraiuolo, and Edward Suh. Timing Channel Protection for a Shared Memory Controller. In *Proceedings of the 20th International Symposium on High Performance Computer Architecture*, 2014.

[91] Yao Wang and Edward Suh. Efficient timing channel protection for on-chip networks. In *Proceedings of the 6th ACM/IEEE International Symposium on Networks-on-Chip.*, NOCS, 2012.

[92] Zhenghong Wang and Ruby B. Lee. Covert and side channels due to processor architecture. ACSAC '06.

[93] Zhenghong Wang and Ruby B. Lee. New Cache Designs for Thwarting Software Cache-based Side Channel Attacks. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, 2007.

[94] Hassan M. G. Wassel, Ying Gao, Jason K. Oberg, Ted Huffmire, Ryan Kastner, Frederic T. Chong, and Timothy Sherwood. `SurfNoC: A Low Latency and Provably Non-Interfering Approach to Secure Networks-on-chip`. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013.

[95] Robert N. M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav H. Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey D. Son, and Munraj Vadera. Cheri: A hybrid capability-system architecture for scalable software compartmentalization. *2015 IEEE Symposium on Security and Privacy*, pages 20–37, 2015.

[96] Emmett Witchel, Josh Cates, and Krste Asanović. Mondrian Memory Protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, 2002.

[97] Rafal Wojtczuk and Joanna Rutkowska. Attacking SMM Memory via Intel CPU Cache Poisoning, 2009.

[98] Rafal Wojtczuk and Joanna Rutkowska. Following the White Rabbit: Software Attacks Against Intel VT-d Technology, 2011.

[99] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. `The SPLASH-2 Programs: Characterization and Methodological Considerations`. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, 1995.

[100] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. The CHERI Capability Model: Revisiting RISC in an Age of Risk. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, 2014.

[101] Steve Zdancewic and Andrew C. Myers. Robust declassification. CSFW '01, 2001.

[102] Steve Zdancewic and Andrew C. Myers. Observational determinism for concurrent program security. In *CSFW*, 2003.

[103] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Secure program partitioning. *ACM Transactions on Computer Systems*, 20(3):283–328, August 2002.

[104] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in histar. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, 2006.

[105] Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazières. Securing distributed systems with information flow control. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, 2008.

[106] Nickolai Zeldovich, Hari Kannan, Michael Dalton, and Christos Kozyrakis. Hardware Enforcement of Application Security Policies Using Tagged Memory. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, 2008.

[107] Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. `Predictive Mitigation of Timing Channels in Interactive Systems`. In *Proceedings of the 18th ACM conference on Computer and communications security*, 2007.

[108] Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. Language-based control and mitigation of timing channels. PLDI '12, 2012.

[109] Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. A hardware design language for efficient control of timing channels. Technical Report http://hdl.handle.net/1813/36274, Cornell University, 2014.

[110] Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. A Hardware Design Language for Timing-Sensitive Information-Flow Security. In *ASPLOS*, 2015.

[111] Yinqian Zhang, Ari Juels, Alina Oprea, and Michael K. Reiter. `HomeAlone: Co-residency Detection in the Cloud via Side-Channel Analysis`. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, 2011.

[112] Lantian Zheng and Andrew C. Myers. Dynamic security labels and static information flow control. *International Journal of Information Security*, 6(2–3), March 2007.

[113] Lantien Zheng and Andrew C. Myers. Dynamic Security Labels and Static Information Flow Control. In *IJIS*, 2007.